REAL-TIME COMPUTER CONTROL OF A HIGH-PERFORMANCE, SERIES ELASTIC, ARTICULATED JUMPING LEG

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree of Bachelor of Science with Honors

at The Ohio State University

By

Simon Curran

* * * * *

The Ohio State University

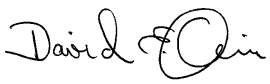
2005

Honors Examination Committee:

Dr. David E. Orin

Dr. James P. Schmiedeler

Approved by:



Adviser
Electrical and Computer Engineering

CONTENTS

Page

ABSTRACT

A lightweight, small-scale, prototype mechanical leg was developed with the intent of being used in a quadrupedal machine capable of a planar gallop. In contrast with wheeled vehicles, robotic quadrupeds do not require a clear path. With only discrete foot holds, they could traverse formidable terrain at speeds once thought possible only by biological systems. Nevertheless, dynamic locomotion requires both explosive leg power and computationally intensive real-time computer control. Research has show that size, weight, computer and actuation power are all issues that must be further explored for robotic galloping to be successfully achieved. The leg studied has two axes of control, one at the hip, and the other at the knee. Each axis is a revolute joint with a single degree of freedom and is actuated solely by a DC motor through a cable and pulley transmission. Series elastic actuation is implemented at the knee to control the shank and seeks to mimic the tendons and muscles found in biological systems. The goal of this project was to establish real-time coordinated control of the two axes using an assortment of lightweight, compact, cutting-edge electronics. A software interface was developed using a C Linux API and combined with efficient coding techniques to achieve real-time supervisory control of each motor controller. As a final demonstration, the leg performed a standing jump whereby it exhibited a high degree of performance so that it may be used in a quadrupedal machine capable of a planar gallop. In this thesis, the design methodology for both the hardware and software will be thoroughly discussed. Particular attention is paid in presenting the electrical components and their associated interfaces. A discussion of the significance of the experimental results is also presented, as well as the expected future research potential for the leg.

ACKNOWLEDGMENTS

I would like to express my sincere thanks to all of those who helped me throughout the course of this project. First and foremost, I would like to thank Dr. David Orin and Dr. James Schmiedeler for their thoughtful guidance and for giving me the opportunity to become part of the multi-disciplinary robotics research group. This past year has been the most rewarding experience for me as an undergraduate and has sparked inside of me, a newfound interest in legged robotics.

Darren Krasny, your counsel on this project and other matters, was without an equal. In only a year's time I feel like I have gained a lifetime of knowledge, thank you. There will no doubt be a large void to fill once you graduate. I would also like to give special thanks to Luther Palmer and Joe Remic. Luther, your practical advice with the hardware saved me countless hours and helped me to get results in time. Joe Remic, without your mechanical wherewithal we would never have had a mechanical leg in the first place. In addition, the overall success this project had at the Denman Undergraduate Research Forum can certainly be attributed to the time sacrifices you made to help me set up this project at the event.

Finally, I would like to thank my parents for their love and support over the years. Especially during the last few months of my undergraduate career, which were the most trying of all. Without my mother's proof-reading and recommendations I would not have been able to complete this thesis on time.

I dedicate all of the time I spent on this thesis and the entire project to Cassandra Winters.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

This thesis focuses on the establishment of coordinated control of a series elastic articulated jumping leg. The leg (Figure 1.1) was developed by Joseph Remic III [1] under the supervision of Dr. James P. Schmiedeler of the Department of Mechanical Engineering at The Ohio State University. The leg was designed to study the use of a lightweight, small-scale, prototype leg with series elastic actuation in a quadrupedal machine capable of a planar gallop (described shortly). Over the past several years, the project has developed into a joint research effort between the Department of Mechanical Engineering at Stanford University, as well as the Department of Mechanical Engineering and the Department of Electrical and Computer Engineering at The Ohio State University. Dr. David Orin of the Department of Electrical and Computer Engineering at The Ohio State University is supervising the integration of the control hardware and software.

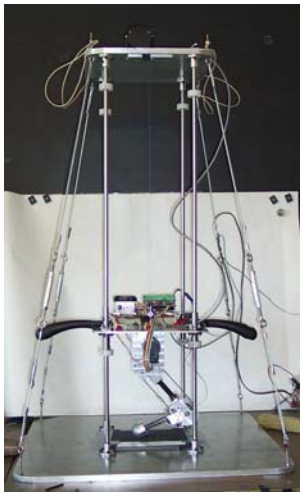


Figure 1.1: Articulated Jumping Leg

Creation of a galloping machine can lead to the development of radically different forms of transportation. In contrast with wheeled vehicles, robotic quadrupeds do not require a clear path. With only discrete foot holds, they could traverse formidable terrain at speeds once thought possible only by biological systems. Their ability to dynamically maneuver, even at high-speeds, would set them apart from any conventional vehicle to date. Furthermore, a small lightweight robotic quadruped could be used in humanitarian removal of land-mines, military reconnaissance and as a personal assist dog. Its ability to be outfitted with a host of sensory equipment would make it an ideal machine in dangerous search and rescue conditions where toxic gases or extreme heat prevent humans or animals from working.

To this end, progress must be made, first, to establish a quadruped capable of a planar gallop. This simplified machine would not require abduction or adduction at the hip because external support would be provided to prevent the machine from rolling. Furthermore, the pronation, supernation, and flexion of an ankle joint is neglected so that research can focus solely on the active articulation of the larger single-degree-of-freedom revolute hip and knee joints. The system does not have any passive joints; however, it does employ series elastic actuation about the knee to more closely mimic the tendons and muscles found in biological systems (see Figure 1.2).

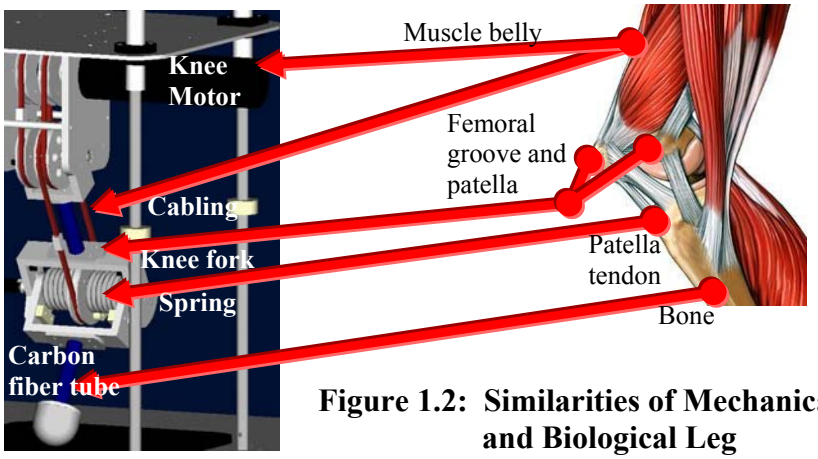


Figure 1.2: Similarities of Mechanical Leg and Biological Leg

To explain further, passive joints are not explicitly actuated and controlled (i.e., there is no motor driving the joint nor is there an associated controller). In the case of series elastic actuation, the joint is actively controlled, but the driving force is transmitted through a passive spring to the joint. A basic configuration of a series elastic actuator is given in Figure 1.3.

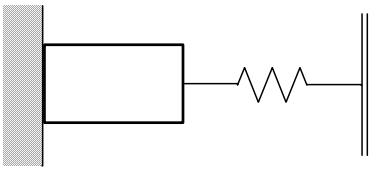


Figure 1.3: Block Diagram for Generic Series Elastic Actuator

The effects of introducing series elastic actuation into the leg's knee control has yet to be fully explored. However, it is known that the passive spring element serves to store energy which is paramount in jumping legged robots [2]. At the same time, its use differs with a traditional robotic rule of thumb that "stiffer is better" because increased stiffness improves the precision, stability, and bandwidth of position control. On the other hand, there is evidence that reducing interface stiffness not only adds the capacity to store energy, it also offers greater shock tolerance, lower reflected inertia, more accurate and stable force control, and can be less damaging to the environment and system [3]. Legged systems, specifically those which see high impact forces at ground contact, can take advantage of these characteristics. In particular, the ability of series elasticity to low-pass filter impulsive shocks greatly reduces peak gear forces and can effectively increase the lifespan of the actuators. Nevertheless, when performing closed loop control in this project the issue of instability introduced by the elasticity must be kept in close consideration.

The prototype leg of this project was designed with the premise that it would be of high enough performance that it could be applied to a future quadruped capable of a planar galloping gaits. Work by Raibert shows that the results from machines with a single leg correlate directly to multiple legged systems [2]. For reasons discussed in the next section, the leg was also specifically designed to be small and lightweight. In general, the quadruped will be no larger than a small dog. The leg's overall performance will be judged by its ability to perform a vertical jump. As depicted in Figure 1.4, the thigh and shank are mounted to an 8" x 10" plate and controlled by cables. The mounted system is constrained by four rails to only allow for vertical motion. This work focuses on establishing coordinated control of each axis of the articulated leg through a blend of electronics and control software.

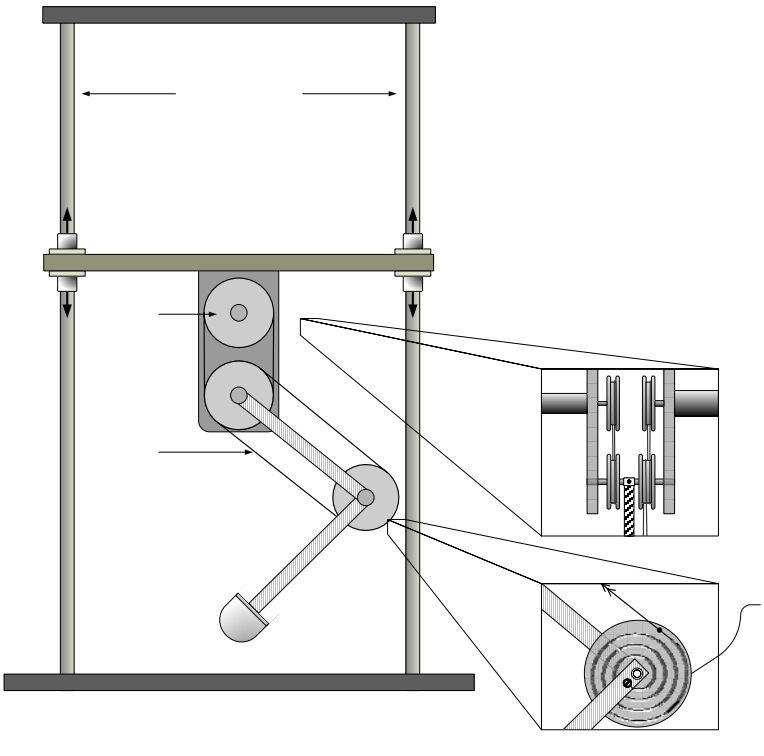


Figure 1.4: Diagram of Vertical Hopper

1.2 Previous Research

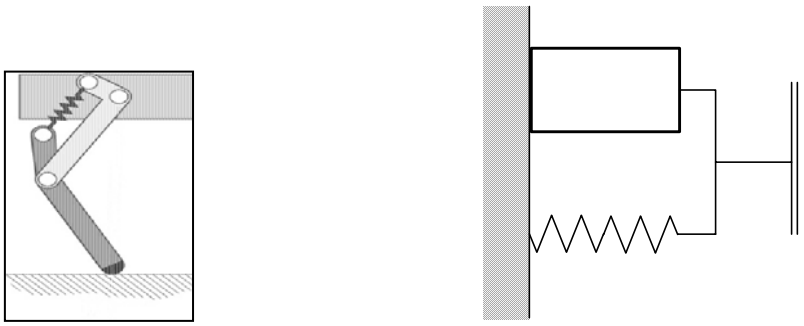
The allure of high-speed locomotion has motivated the design of many legged quadrupedal robots over the past several decades. While there is no lack of robots capable of walking (e.g., the OSU Hexapod [4], the Adaptive Suspension Vehicle [5], and Case Western Reserve University's Robots I, II, and III [6, 7]) there are no known robotic systems capable of achieving a well-controlled, biological gallop. However, credit should be given to Smith and Poulakakis [8] who were able to demonstrate what appears to be the first rotary gallop in the quadruped robot Scout II. Nevertheless, some of the defining features of a biological gallop were not present in Scout II. For one, the robot only moved in a tight circular trajectory and did not exhibit heading control, which is one of the fundamental features of biological locomotion [9].

While little is known why the gallop is the preferred high speed running gait of most cursorial animals, there are several prevailing theories. One argues that the gallop is simply more energy efficient at higher speeds [10]-[11], while others believe that the characteristic smoothness of the gait results in lower peak forces on the legs [12]-[14]. Schmiedeler [15] reasoned that combined, the two factors could explain the attractiveness in terms of energy efficiency. Regardless, galloping is an efficient, effective, and robust means of high-speed locomotion for many biological quadrupeds – yet it remains an elusive goal in the field of legged robotics.

A quadruped, developed by Raibert in the early 1980's, used pneumatic and hydraulic actuators, with prismatic legs, to successfully trot, pace, and bound [16]. More recently, work on the six-legged robot Rhex has shown that compliant legs can walk very

efficiently across uneven surfaces and has sparked an interest in adding compliance to articulated legs.

However, size and weight have become a major consideration in developing a dynamically stable galloping quadruped with articulated legs. While Stanford University has explored many forms of actuating the articulated legs on their large quadruped named KOLT [17], they are still severely hampered by the machine's weight. In addition to the difficulties with Raibert's current quadruped, named Big Dog, it is becoming evident that size and weight are issues that must be further explored for robotic galloping to be successful. Not only does the heavy quadruped need high amounts of actuation energy, it also experiences large impact forces with the ground that require the continual maintenance and replacement of parts. For example, KOLT must make use of large springs in a parallel arrangement (see Figure 1.5 (a)) to store energy during stance. However, during flight the springs actually oppose the actuators, further increasing power requirements (see Figure 1.5 (b)). In addition, past and present research has shown that controlling locomotion of a quadruped during a gallop is a computationally intensive process [9], [18]-[20].



(a) (b)

Figure 1.5: (a) Simplified Leg from KOLT Showing a Parallel Spring Arrangement, and (b) Block Diagram for a Generic Actuator with a Parallel Spring Arrangement

On the other hand, recent breakthroughs with evolutionary search algorithms at The Ohio State University have shown, in simulation, that a quadrupedal machine outfitted with light-weight, articulated legs with series elastic actuation is capable of producing a high-speed, dynamically stable gallop [9]. The leg used in Krasny and Orin's simulation, albeit lighter, is still very similar to the robotic leg designed by Remic [1] and used in this thesis.

1.3 Research Objectives

The main objective of this work is to continue the efforts of creating a galloping quadruped by developing the controls for a powerful, lightweight, articulated jumping leg. This includes the complete integration of an embedded system, DC motor actuators, amplifiers and interface circuitry with the mechanical system. Upon completion, software must be implemented on the embedded system to establish real-time computer control of the two actuators. Finally, in order to measure the performance of the system, a control routine must be developed to coordinate the hip and knee joints through a complete vertical jump.

Completion of the hardware required purchasing two light, compact, and powerful DC brushless motor amplifiers from MSK Corporation and a set of three phase brushless DC motors from Maxon Motors. A state-of-the-art, 400MHz credit card sized embedded system by K-Team was chosen because it offered a Linux platform with four integrated motor controllers and robust I/O in a compact light-weight package. Significant testing was needed in order to interface and operate the eclectic mix of electronics.

The establishment of real-time control software on the embedded system involved developing a C application using K-Team's application programming interface (API). The API provides high level C commands for accessing the motor controllers on the KoreMotor board from an application running on the KoreBot board. Unfortunately, the commands each have an associated time delay, making real-time control a non-trivial task. Furthermore, the embedded system was new enough that its accompanying documentation was incomplete, and provided for a challenging development experience.

The final objective of this thesis project was to develop a state based control application that coordinated the hip and knee motors to produce a jump. This first required tuning the proportional, integral, and derivative (PID) controllers for each axis, as well as effectively sensing ground contact. It was desired that the jump routine transition through Stance, Crouch, Thrust, and Flight / Touchdown states. In this way it was possible to compare the performance of the jump with that of a simulation produced by Krasny.

1.4 Thesis Organization

Chapter 1 of this thesis presents background information on legged locomotion, and, in particular, dynamically galloping quadrupeds. Key features of the leg used in this project were discussed in some detail. It was noted that the mechanical design was produced by Joseph Remic as part of his graduate work. Next, a summary of prior work on legged robots was presented and the motivation for building a light weight, small-scale, articulated leg was discussed. Finally, a summary of the research objectives of this project was presented.

Chapter II will focus on the hardware integration for the robotic leg. The various components of the system will be discussed, including the amplifiers, motors, optical encoders, interface circuitry, and power supplies. In addition, the main control connections for each axis will be summarized. Finally, special emphasis will be placed on explaining the characteristics of the motor drive system with an in-depth look at the internal architecture of the MSK 4360 amplifier.

Chapter III will discuss both the control aspects, and software development of the KoreBot and KoreMotor boards. In particular, a general overview of the control architecture will be given, with a more detailed look at how real-time demands can be met. Then, a look at the software development will outline what is needed to write and compile an application for the KoreBot Linux platform. Finally, a discussion of the LibKoreBot C API will be presented, including a discussion of several important supervisory and control commands. These will become important in Chapter IV, when the details of the control code for the jumping leg are presented.

Chapter IV will present the most pertinent results of this project. Specifically, emphasis is placed on the ability of the leg to achieve a high performance jump. Special attention will also be paid to the state based software responsible for producing the jump. Then, there is an important discussion on how to tune the PID controllers and prevent overheating of the motors. A broad discussion of the final electronics package will also be given.

Finally, Chapter V is a summary with conclusions of the research performed in this project. It also provides recommendations for future study and applications.

CHAPTER 2

EMBEDDED SYSTEM, MOTOR DRIVE, AND INTERFACING

2.1 Introduction

The overall success of the jumping leg depended on meeting low weight restrictions while at the same time providing substantial power to directly thrust the leg into a vertical jump. Without adding heavy springs or hydraulics, the power stroke needed to jump was provided solely by high performance direct current (DC) motors. Also, traditional control hardware was abandoned for smaller state-of-the-art controllers and amplifiers. At the same time, considerations had to be made as to the ease of which the components could be replicated and used on a quadrupedal galloping machine.

Controlling the locomotion of a quadruped during a gallop is a computationally intensive process [18]-[20]. Therefore, if the jumping leg is to be integrated into a future quadruped, it must have enough processing power to handle the real-time demand of calculating leg trajectories during all phases of locomotion. A state-of-the-art, 400MHz credit card sized embedded system by K-Team was chosen because it offered a Linux platform with integrated motor controllers and robust I/O in a compact light-weight package. Unfortunately, the embedded system was new enough that its accompanying documentation was incomplete. This introduced additional complexity and required continual contact with K-Team.

The drive system responsible for articulating both the thigh and knee axis require a set of amplifiers and DC motors. Simulation set the precedent for torque, speed, and

weight requirements. To this end, two three-phase brushless DC motors with high specific power were chosen from Maxon Motors that meet the requirements. The details of the three-phase brushless DC motors are discussed in Section 2.5.

In a continuing effort to keep weight to a minimum, two bare-bone brushless motor amplifiers were chosen to drive the DC motors. Built for aerospace applications, the amplifiers were some of the most powerful, compact, and light-weight models available on the market. Nonetheless, the amplifiers introduced additional complexity in that they required supporting electronics as well as a host of interface circuitry. By far the most demanding task of the electronics was in designing and implementing an interface that successfully combined all the high-performance electronic components.

2.2 System Overview

The system diagram of Figure 2.1 emphasizes the important electrical systems and how they interface with the mechanical design. The electronics serve as a means to control and measure the physical system. The sensory information can also be combined to create an adequate kinematics model of the leg for study.

2.2.1 Mechanical System Overview

Again, the leg has two single-degree-of-freedom revolute joints, one at the thigh and the other at the knee. As the diagram depicts, the thigh motor uses a cable to directly drive the thigh axis through a pair of equal sized pulleys. Therefore, relatively speaking, the angle that the thigh makes with the vertical, θ_t, is the same as that of the thigh motors

output, θ_{tm}. On the other hand, the shank connected at the knee is not directly driven by the knee actuator. As the diagram illustrates, the drive cable from the knee motor connects to a freely mounted torsional spring about the knee axes. The opposite end of the torsional spring connects directly to the shank and is solely responsible for transferring cable forces to the shank. In this regard, the torsional spring can be considered to be in series with the knee motor and the shank. As the cabling that travels from the knee motor to the knee axis must travel around the hip axis, the knee and hip can be considered to be a coupled system. Control of this system is left for a later chapter but the various physical parameters are detailed here.

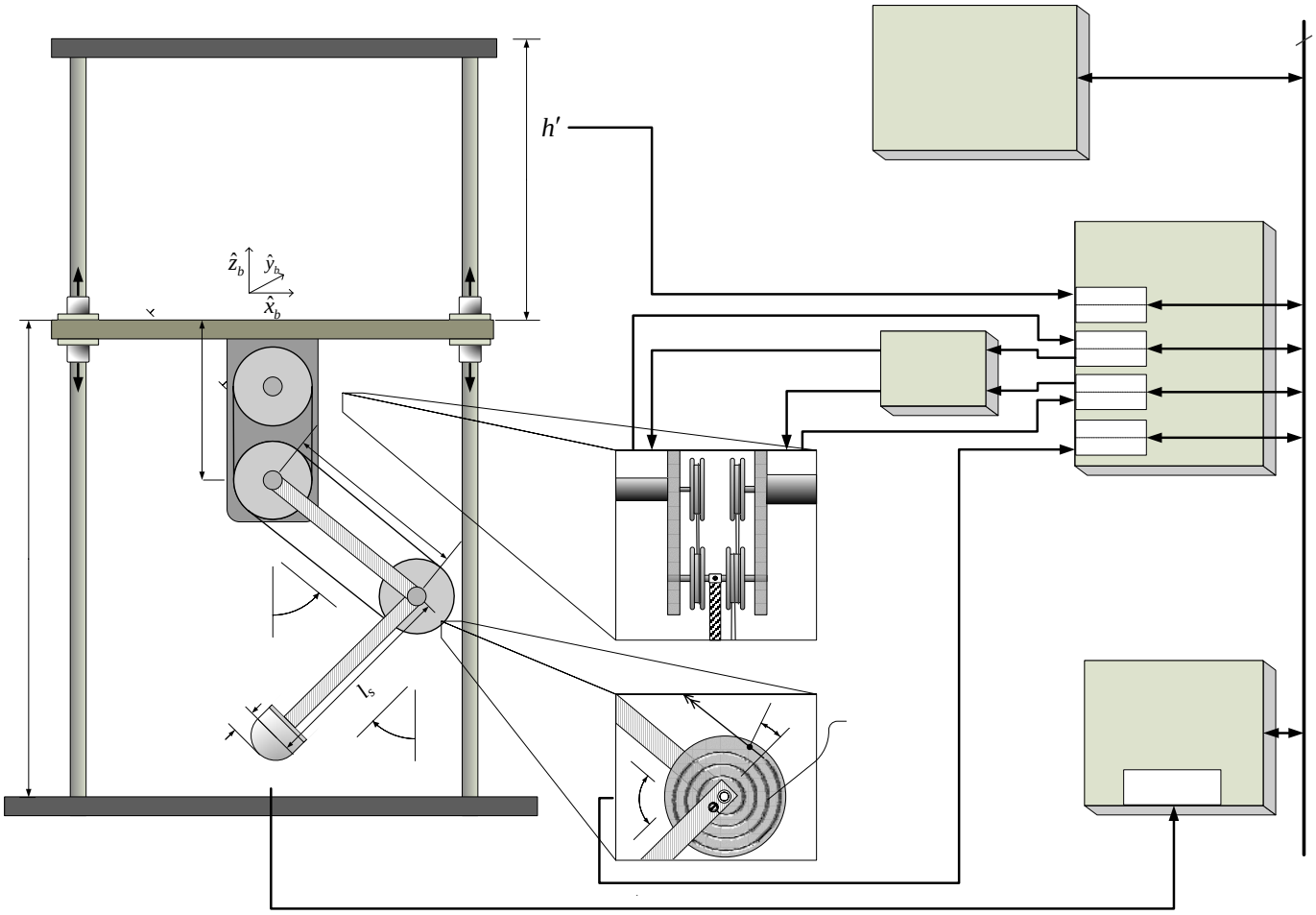


Figure 2.1: Embedded Microprocessor System Interface Diagram

The angle that the shank makes with the vertical is designated θ_s, and is the sum of both the output of the knee actuator, θ_{sm}, and the angular displacement in the torsional spring, θ_{ds}. Another angle at the knee provides the angle of the shank relative to the thigh and is designated as θ_k because it is, in actuality, the knee angle.

The entire leg is mounted to the bottom of a square plate with the thigh axis fixed a distance, l_h, below the plate. The plate is constrained by four rails to allow only vertical motion. The thigh and shank lengths are also fixed at l_t and l_s respectively. The vertical distance between the moving plate and the stationary top plate is designated by the variable h', as it can be used to determine the height, h, of the leg. Finally, a micro switch is embedded in the foot and indicates whether or not the foot is in contact with the ground. The two states of contact are designated by the variable σ where a '1' indicates ground contact and a '0' indicates flight. For a list of values for the physical parameters outline in this section please see Appendix A1.

2.2.2 Electrical System Overview

The electrical system serves to control and monitor the mechanical system. The electro-mechanical crossover ultimately resides with the DC motors. Sensors are used as both feedback devices for control, as well as monitoring devices to study the overall behavior of the physical system. With the exception of a 48 volt direct current (VDC) power supply, all electronics are compacted onto a perforated board and mounted on the moving plate of Figure 2.1. Separate DC to DC converters also exist on the perforated board to serve the embedded system and interface electronics.

A total of five sensors interface with the physical system. Two digital encoders are attached to the thigh and knee motors to measure θ_{tm} and θ_{sm} respectively. A third, small lightweight Gurley encoder is fixed about the knee axis to measure θ_k. A string encoder is fixed to the top plate and uses a small string to measure h', the distance between the top plate and the moving plate. The last sensor is simply a micro switch embedded in the foot and is on or off depending on whether the foot is in contact with the ground.

The embedded system is composed of three boards, each serving a specific purpose. The KoreBot board contains the embedded Linux platform and acts as the supervisory controller board. The KoreMotor board has four independent motor controllers used for both control and data acquisition. Finally, the KoreIO board hosts robust digital and analog I/O but is mainly used as a digital input. The three boards are stacked together using KB-250 modules, allowing the boards to share, amongst other features, both power and an inter-integrated circuit (I2C) bus for communication.

Two of the motor controllers on the KoreMotor board are called upon to control the thigh and knee motors. They do not, however, directly connect to the motors, but instead pass through the motor drive system. The motor drive system consists of the motor amplifiers and interface electronics necessary to actuate the DC motors. The remaining two motor controllers on the KoreMotor board are only used to accept input from the two digital encoders measuring h' and θ_k. By combining the sensory information with important physical parameters outlined in the previous section, a complete kinematics model of the leg is possible.

2.3 Motor Drive and Interfacing

The electrical connections for each joint of the leg follow the same general conventions presented in this section. For a complete electrical system diagram see Appendix A2. Figure 2.2 below shows the connections for a single axis, controlled by just one controller on the KoreMotor board. The only difference between thigh and knee electronics are a few resistors that set the gain in the filter circuit. For the sake of clarity, only the connections of interest are shown in the diagram. As mentioned before, the KB-250 module provides both power as well as a two-wire I2C bus for inter-board communication. Unfortunately, because of the eclectic mix of hardware no standard cable connection existed. Instead, custom wiring was created.

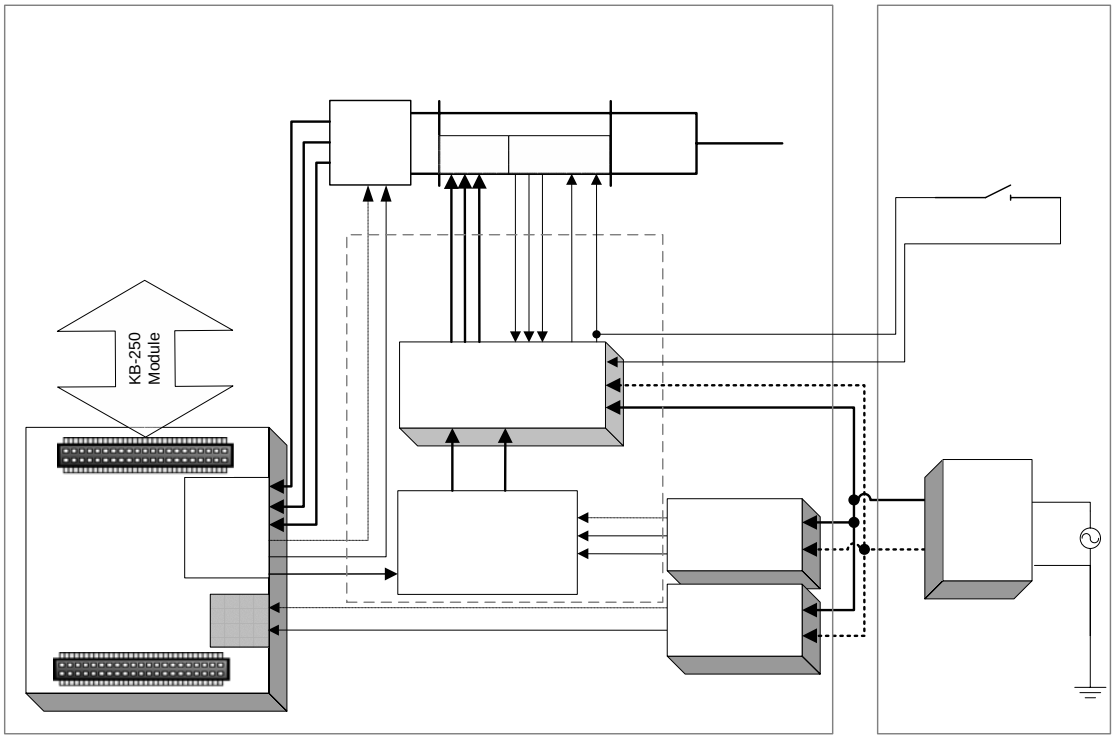


Figure 2.2: Single-Joint Motor Drive and Interfacing

Beginning with the right end of the diagram, the 48VDC/900Watt regulated power source resides off of the moving system and provides power to all onboard electronics through a heavy gauge wire. Two DC to DC converters step down the 48VDC bus to power other electronics components at +5VDC and +/- 12VDC. The 48VDC is in reference to "GND1", and the lower voltages are referenced to "GND2."

The amplifiers power the DC motors with the 48 volt supply through a three-phase connection. The three phases of the amplifier, A, B, C, are connected to motor windings 1, 2, and 3, respectively. An off-board disable switch is connected to the amplifier and can turn on or off the amplifier's motor drive. The amplifiers also provide power to the Hall sensors, whose outputs are routed back to the amplifier as a feedback signal for the brushless commutation. In actuality, the amplifier is itself a current controller with an inner control loop, and drives the motor with a current proportional to the DC voltage applied across the "ICommand" pins. This will be discussed in greater detail in the following section.

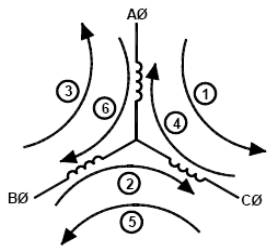
Each motor is equipped with a Hewlett-Packard optical encoder, which provides information on the relative position of the rotor, θ_{tm} and θ_{sm}. This data is fed directly to a motor controller on the KoreMotor board and is the main feedback signal used to sustain PID control of each joint. The signal labeled "pwmCMD" is a +5 V pulse width modulated (PWM) voltage with reference to "GND2" and is output by the motor controller with a duty factor proportional to the desired command. As mentioned before, the motor amplifier cannot accept the modulated command signal, so the "pwmCMD" must first be filtered by the low-pass filter to extract the DC component. A later section is devoted to a discussion of the low-pass filter circuitry.

2.4 Motor Amplifiers

Two compact 4360-series PWM brushless DC motor controllers were purchased by M.S. Kennedy Corporation (MSK) to power the thigh and knee actuators. Each unit weighs only 44 grams and can provide 10 amps of output current at 55 VDC. The 20 pin amplifiers are electrically isolated in a small hermetic package and have good thermal conductivity allowing for easy heat sinking. Included in the package is all the bridge drive circuitry, hall sensing circuitry, commutation circuitry and all the current sensing and analog circuitry necessary for closed loop current mode (torque) control. As a result, the amplifier can maintain motor current at exactly twice that of the voltage provided on the "ICommand" input. For a detailed summery of the MSK4360 connections and specifications see Appendix A3.

2.4.1 Amplifier Characteristics

Before using the brushless DC motor (BLDC) torque amplifier it is necessary to first discuss some of its important characteristics. As stated before, a BLDC controller uses hall devices to sense the rotor position inside the motor. There are six steps that a three phase BLDC motor commutes through per "electrical revolution," and depending on the number of poles in a given motor there may be more electrical revolutions needed for a single mechanical revolution. As Figure 2.3(a) depicts, in any one of the commutation steps, only two of the three motor phases are energized. Figure 2.3(b) shows the commutation truth table where each row corresponds to a step of Figure 2.3(a).

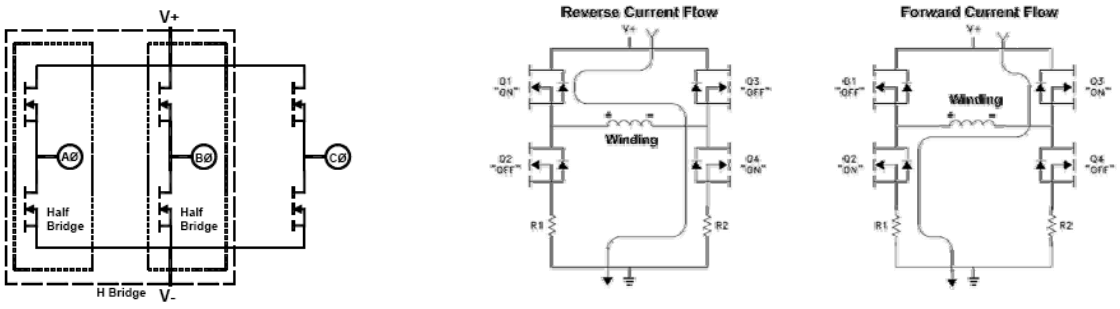


HALL SENSOR PHASING 120°			ICOMMAND = POS.			ICOMMAND = NEG.		
HALL A	HALL B	HALL C	AØ	BØ	CØ	AØ	BØ	CØ
1	0	0	H	-	L	L	-	H
1	1	0	-	H	L	-	L	H
0	1	0	L	H	-	H	L	-
0	1	1	L	-	H	H	-	L
0	0	1	-	L	H	-	H	L
1	0	1	H	L	-	L	H	-

(a) (b)

Figure 2.3 (a) Commutation Steps and (b) Truth Table for a 3-Phase Brushless DC (BLDC) Motor

To activate a pair of windings, the supply voltage is switched on across the windings causing current to flow in one direction. As Figure 2.4 (a) illustrates, MSK employs an H-bridge of metal-oxide semiconductor field-effect transistors (MOSFETs) to switch the voltage as necessary. A total of three H-bridges are present in each amplifier, yet Figure 2.4 (b) shows that only two are necessary to energize a pair of motor windings. Figure 2.4 (b), also shows that when the single transistor in the top left half bridge, Q1, switches V+ across phase A, and the lower right transistor in the other half bridge, Q4, switches V- across phase B, current flows forward through the set of windings. However, if transistors Q2 and Q3 are switched on and the previous pair switched off, current flows in the reverse direction.



(a) (b)

Figure 2.4: (a) The Three H-Bridge MOSFETs Used by MSK Amplifier, and (b) Current Flow Through H-Bridge and Motor Windings

It is the switching of the supply voltage by the transistors that ultimately provides the motor with a PWM signal. The most important factor of the PWM signal is the duty cycle. Depicted in Figure 2.5, the duty cycle is a measure of the amount of time the pulse is on. The electrical characteristics of the motor smooth the voltage spikes, thus creating a direct relationship between the PWM duty cycle and the power supplied to the motor. Further discussion of the motor's electrical characteristics is left for a later section.

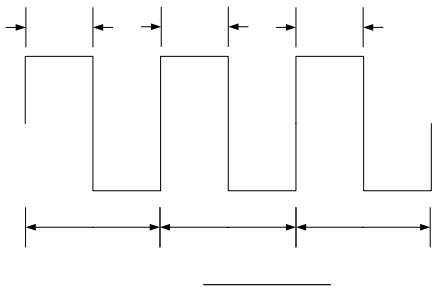


Figure 2.5: Characteristics of a PWM Signal

In general, BLDC motors have four modes or quadrants of operation. These modes are best explained when viewing the torque vs. speed plot of Figure 2.6. In quadrants I and III, for example, the motor is spinning in the same direction as the applied torque. In quadrants II and IV, however, motor torque is being applied opposite to the motor's speed. In the second case, torque is being used to brake the motor, with the motor generating power as a result. Both modes of control can be implemented using the set of H-bridges in Figure 2.4 (b) and only differ in their

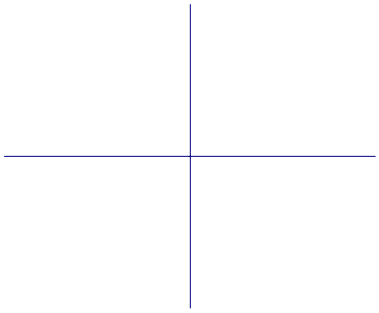


Figure 2.6: Four Quadrants of Motor Operation

modulation of the four switches. In robotics, where a high bandwidth servo loop is necessary, all four quadrants of operation should be possible by a motor controller and is provided by the MSK 4360.

An important concept that is worth noting are the two methods of controlling torque when the motor is at zero speed. In order to maintain control of the current loop at this point, it is imperative to be constantly sensing the current through the windings. However, if the PWM signal is simply modulating the voltage of a winding on and off, then as the command voltage decreases so does the current through the windings, as seen in Figure 2.7. As the command is decreased to zero, the duty cycle of the PWM signal becomes zero, and there is no longer any current through the sensing resistors R1 or R2 in Figure 2.4 (b) to provide feedback for the controller. This discontinuity results in a loss of control and is not acceptable in many applications.

Figure 2.7: Voltage and Current Waveforms Near Zero Current Command

The MSK 4360 provides a different method of controlling torque at zero speed that allows for constant sensing of winding current, and creates a stable controller. At zero current command, the 4360 actually modulates the voltage at a 50% duty cycle with each half cycle reversing polarity across the windings. A 16 KHz timer creates a total cycle time of about 63 µs. The average or DC value of such a command is in fact zero.

Figure 2.8: Voltage and Current Waveforms of Zero Current Command for MSK Amplifiers

Note that modulating the polarity here gives, in effect, double the supply voltage or double the bandwidth as the previous method. Again, motor characteristics cause the square PWM voltage wave to create a triangular current wave, the net result of which is zero current and zero torque. Yet with current always flowing in the windings, both resistors of Figure 2.4 (b) will be providing feedback at all times, and the controller will not go out of control. While this provides superior control bandwidth at zero torque, it also causes heating issues in the motor that will be discussed later.

2.4.2 Electrical Setup of Amplifier

The MSK 4360 is a bare-bones amplifier and requires certain external electronics before it can be operated. While this adds another level of complexity to the system, it also makes for a lighter than traditional amplifier, as only the necessary electronics are implemented. The amplifier's pins are neither spaced nor sized to fit into a standard socket. Individual gold-plated pin sockets were slid onto each pin, providing a solder point, instead of using the pin itself. Careful attention is a must when working with the closely packed pins so that no connection is inadvertently shorted out.

First, with all of the internal switching taking place in the amplifier, it is imperative that good ground planes are established. The amplifier has a separate internal 15 V switching mode supply to operate its analog control logic. Both this voltage and the main 48 V motor voltage get separate 2" x 3" ground planes (SIGGND and V+ RTN, respectively) and are tied together through a single thick connection. Having separate ground planes helps to reduce the interference or "cross talk" caused by having two separate switching sources that could otherwise render the amplifier useless. In general, the ground planes aid by absorbing the induced and ambient electromagnetic noise by providing a large flat surface to dissipate any ground potential rises. Nonetheless, the ground planes alone do not provide adequate noise immunity.

Various capacitors are added to aid in suppressing noise transients as well as regulate the voltages. A 0.1uF capacitor was placed between both the +15 V and -15V pins and ground to damp the high frequency noise. Also, 10uF tantalum capacitors were needed at the same locations to help regulate the voltage. In addition, a small switching mode inductor capable of running at 250 KHz and about 1 amp DC was needed for the internal DC to DC converter. A suitable model was found and purchased from Coilcraft.

A similar approach of using bypass capacitors was taken for the 48 V main supply bus, however, more care was needed in laying out this aspect of the system. Every time a MOSFET switches, it introduces a high frequency voltage spike that appears on top of both the bus voltage and the back electromagnetic fields (EMF) of the motor. In excess these spikes can destroy the controller's H-bridge or even the capacitors meant to suppress them. It is imperative to place the bypass capacitor as close to the amplifier as possible. A 10uF polymer capacitor was chosen to serve this purpose.

At the same time, large voltage dips caused by sudden high current commands must be avoided. Besides the resulting performance reduction of the motor, the MOSFETS experience more heating when the voltage dips. The issue of heating is discussed in the next section. Even though the main bus is regulated by the power supply, localized dips in voltage must be reduced by placing a 2400uF bulk capacitor across V+ and ground near the two amplifiers.

The final step in setting up the 4360's external electronics is creating the controller's current compensation circuit with a resistor and two capacitors. As Figure 2.9 shows, the circuit is like that of a typical controller block of any feedback system.

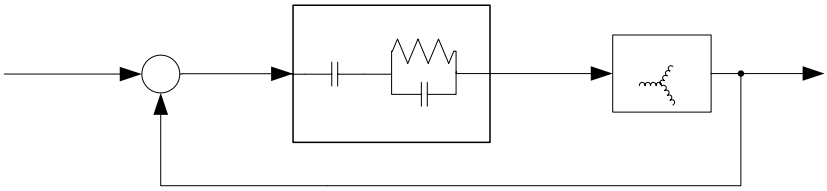


Figure 2.9: MSK Motor Amplifier's Current Loop Error Amplifier (E/A) Compensation Network

As a reminder, it should be noted that this feedback loop is only an inner control loop and will always be sufficiently fast enough as to not be of concern to the closed loop control performed by the KoreMotor controller. The RC circuit implements a basic PID controller between the current loop error amplifier inverting input (E/A –) and the current loop error amplifier output (E/A OUT). Luckily MSK provides a generic RC circuit that requires no tuning and remains stable when controlling both Maxon thigh and knee motors. Nevertheless, it should not be assumed that this holds true for all BLDC motors.

If an attached motor ever squeals or whines it is likely that the current controller is out of control and the compensation circuit should be adjusted. Analysis shows that resistor R and capacitor C1 are directly related to the proportional and derivative gains of the control loop. Therefore, the most advisable method would be to experimentally tune the components by giving the amplifier a step command and observing the current monitor pin. A combination of increasing the damping, C1, and decreasing the gain, R, should prove to be successful.

Connecting the amplifiers with the motors is straight forward and was outlined in section 2.3. All possible combinations of motor phases and hall sensors were explored. It was determined that motor drive A, B, and C on the amplifier correspond to winding 1, 2, and 3 on the Maxon motors. Hall Inputs A, B, C also connect to hall sensors 1, 2, and 3 on the motors, with V+ and GND provided by the amplifier's +15 V and SIGGND, respectively. A disable pin is also provided for externally disabling the output bridge. The pin is internally pulled low by a 5K ohm resistor and enables the bridge. An off-board manual switch can connect this pin to transistor to transistor logic (TTL) high (+15V), disabling the motor drive.

2.4.3 Mechanical Setup of Amplifier

Finally, in order to push the amplifiers close to their rated power output, heat sinks were attached to allow for adequate heat dissipation. Most of the amplifier's heat is created by thermal losses in the main junction of six power MOSFETs which are located on the bottom of the amplifier. As a result, the amplifiers were mounted to the perforated

board in a “dead bug” or bottom up configuration so that the heat sink could be provided with sufficient space and ventilation.

The total amount of thermal energy produced by the MOSFET junction was determined as follows. If the junction were running at its maximum rated condition of 10 amps at 150^{O} C, there would be a 1.92 volt drop across the H-bridge. The 1.92 volts of potential energy at 10 amps means 19.2 Watts are dissipated during the “on” portion of the duty cycle. Simulations by Darren Krasny showed that the motor would only be at full power for half a second during steady state jumping, giving an average power dissipation over a one second interval of 9.6Watts. However, during the tuning of controller gains, discussed in a later chapter, the amplifier could easily see maximum rated conditions for longer than a second. For this reason, aluminum finned heat sinks proven to dissipate 25 watts of heat from Intel Pentium III processors were obtained for each amplifier (Figure 2.10).

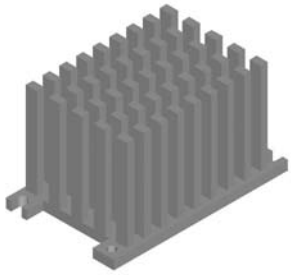


Figure 2.10: Aluminum Finned Cross Convection MSK Heat Sink

The individual heat sinks had to be manufactured to fit the bolt pattern and foot print on the bottom of the amplifiers. Proper heat sinking techniques were followed. The surface of the heat sinks in contact with the amplifiers were flat and smooth to avoid creating insulating air pockets. A thermally conducting compound was spread paper thin between the amplifier and heat sink to provide a solid thermal interface. The two

components were secured together with two bolts diagonally across from each other. Finally, a small 12VDC CPU fan was placed atop each heat sink to optimize the amount of air forced across the aluminum fins and increase heat dissipation another 30 percent.

2.5 Motors and Encoders

The motors that provide thigh and knee actuation are electrically commutated 3-phase brushless DC servomotors with low impedance windings manufactured by Maxon. Referring to Figure 1.1, two HED 5540 optical encoders manufactured by Hewlett-Packard are attached to the thigh and knee motors to measure θ_{tm} and θ_{sm} respectively. A third, small lightweight optical encoder from Gurley is fixed about the knee axis to measure θ_k. Finally, a digital string encoder by Unimeasure is fixed to the top plate and uses a small string to measure h', the distance between the top plate and the moving plate.

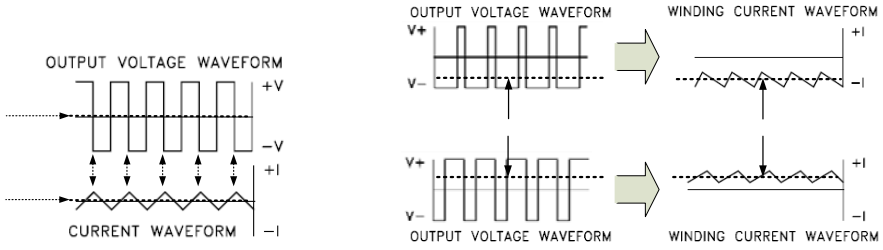
2.5.1 Motors

Maxon motors have proved to be reliable robotic actuators for many years. Model EC 32 was selected for the thigh axis and is rated at 80 watts, but is temporarily overloaded to 160 watts. A slightly larger motor, Model EC 40, was chosen for the knee actuator because it is responsible for delivering the most torque and thereby thrusting the leg into a jump. The knee motor is rated at 120 watts, however, it is briefly pushed to 480 watts while jumping. A summary of the specifications for both of the motors can be found in Appendix A4.

The motor's low impedance windings allow larger currents to flow through the windings, thus creating larger torques. At the same time, the current is predominately

responsible for heating the resistive element of the motor windings. The basic principle for power dissipation due to the copper resistance of the motor windings is I^2R where I is current and R is resistance. The problem of motor heating is perpetuated by both the need to overpower the motor as well as the type of motor amplifier.

As mentioned in the previous section, the amplifier maintains full four quadrant control of the motor by means of continually modulating the polarity of the supply voltage across the windings, even at zero current command. The electrical characteristics of the motor do not need to be discussed in detail, however, it is important to mention that the windings have both resistance and inductance. One of the properties of an inductor is that the current through it cannot change instantly. Therefore, the square voltage waves created by the amplifier's PWM signal cannot create square current waves. Instead, the current takes the form of a triangle wave, with peaks occurring at the transitions of the PWM signal as shown in Figure 2.11 (a). Just as the DC component of the PWM voltage command determines the net voltage, the DC value of the triangular current wave determines the net current through the motor and is proportional to the actual torque produced by the motor (Figure 2.11 (b)).



(a) (b)

Figure 2.11: (a) Production of Triangular Current Waves from Square Voltage Waves, and (b) Creation of Positive and Negative Motor Currents

The motor's thermal characteristics, on the other hand, do not respond in the same fashion. For example, the zero net current depicted in Figure 2.11 (a) does mean that there is zero net heat dissipation. The alternating current (AC), or back and forth current flow about the zero, creates heat in each direction. Recall that the power dissipation is I^2R, therefore current direction does not matter. The combination of low winding resistance and inductance leads to triangular current waves of large amplitude in both the thigh and knee motors. At zero net current the knee motor has a peak-to-peak triangular wave of 5 amps. That current through the 1.69 ohms of winding resistance creates 42 watts of peak energy each PWM cycle and after every second the motor windings release 42 watts of heat. In addition, if there is a net DC current present, that must also be accounted for in determining total heat dissipation. Therefore, when the knee is provided 10 amps of net current the total power dissipated is 10^2 x 1.69 + 42 and equals 211 watts. Table 2.1 summarizes the pertinent information for determining total heat dissipation of both motors.

Table 2.1: Heat Dissipation For Each Motor

Motor	Peak to Peak AC Current (amps)	Winding Resistance, **R** (ohms)	AC Power Dissipation, $\mathbf{P_{AC}}$ (watts)	DC Power Dissipation $\mathbf{P_{DC}}$ (watts)	Total Power Dissipation, $\mathbf{P_{TOT}}$ (watts)
Thigh EC-32	3	5.50	49.5	$\mathbf{I_{NET}^2 * R}$	$\mathbf{P_{AC} + P_{DC}}$
Knee EC-40	5	1.69	42	$\mathbf{I_{NET}^2 * R}$	$\mathbf{P_{AC} + P_{DC}}$

2.5.2 Encoders

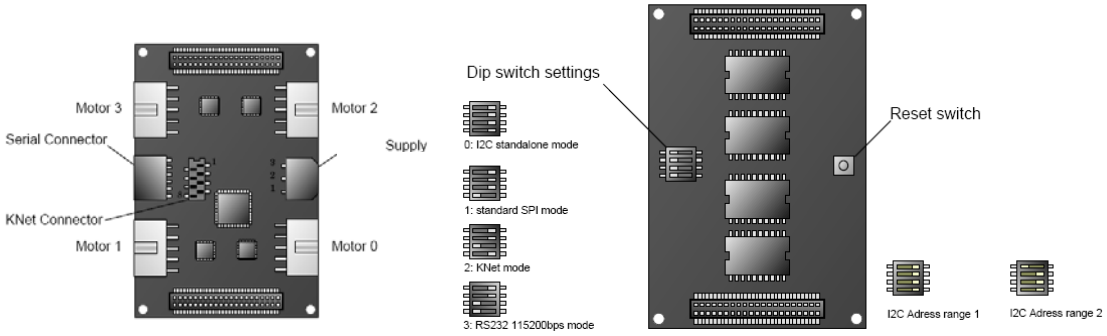
The HED 5540 optical encoders provide high resolution outputs which can be easily interfaced to the KoreMotor board through a set of pull-up resistors. However, no resistors were needed when interfacing the Gurley or Unimeasure encoders with the KoreMotor board. Each encoder has a code wheel with 500 equally spaced apertures around the circumference and an equal number of apertures on the stationary phase plate. A light beam aimed at the code wheel is transmitted only when the apertures on the code wheel and phase plate line up. Thus there are 500 light and dark periods per revolution. Two photodiodes are arranged inside the encoder such that two signals are generated in quadrature (phase difference of 90^{O}). This allows for an effective resolution of 2^2*500, or 2000, counts per revolution. When this resolution is combined with the gear ratio for each motor axis, very high degrees of resolution may be obtained. The cable and pulley drive system add no further gearing to the joints. Table 2.2 summarizes the total resolution for each encoder and Appendix A5 summarizes their required connections.

Table 2.2: Total Resolution for Each Encoder

Parameter	Description	Encoder	Resolution	Gear Ratios	Total Resolution
θ_{tm}	Thigh Motor	HP HED 5540	2000 (counts/rev)	33:1	66,000 (counts/rev)
θ_{sm}	Knee Motor	HP HED 5540	2000 (counts/rev)	66:1	132,000 (counts/rev)
θ_{k}	Knee Angle	Gurley R112	16000 (counts/rev)	N/A	16,000 (counts/rev)
h'	Relative Height	Unimeasure JX-EP	794 (counts/inch)	N/A	794 (counts/inch)

2.6 K-Team Motor Controller Board

The KoreMotor board is the central element in the control system for the robotic leg. As Figure 2.1 showed there are four separate motor controllers and motor ports. Figure 2.12 (a) is an illustration of the physical layout of the ports and PIC micro controllers. A fifth PIC microcontroller manages the boards communication, dispatching orders to the controllers. However, in this project the fifth microcontroller is not required because each motor controller is directly addressable through the I2C bus. Physical layouts of the other K-Team boards can be found in Appendix A6.



(a) (b)

Figure 2.12: (a) Top and (b) Bottom of KoreMotor Board

2.6.1 KoreMotor Interface

Each motor controller and motor port was designed to connect to a single phase brushed DC motor with quadrature encoders. An H-bridge at each port is responsible for creating the "pwmCMD" signal of Figure 2.5 by switching the voltage provided at the "PWM Supply" of Figure 2.12 (a). Motor direction and duty cycle is controlled with only two wires. The PWM line is modulated at 10kHz and has a controllable duty cycle

with 9-bits of resolution. The second line acts as a reference voltage for the PWM voltage, thereby determining motor direction. When this line is set to ground of the PWM Supply (GND2), the duty cycle of the PWM signal remains directly proportional to the voltage supplied to the motor and the effective voltage is positive. However, when the second line is set high, then the "off" part of the PWM's duty cycle effectively determines the amount of negative voltage at the motor. The motor controller always takes this into account and adjusts the duty cycle as necessary. Refer to Figure 2.13 for clarification.

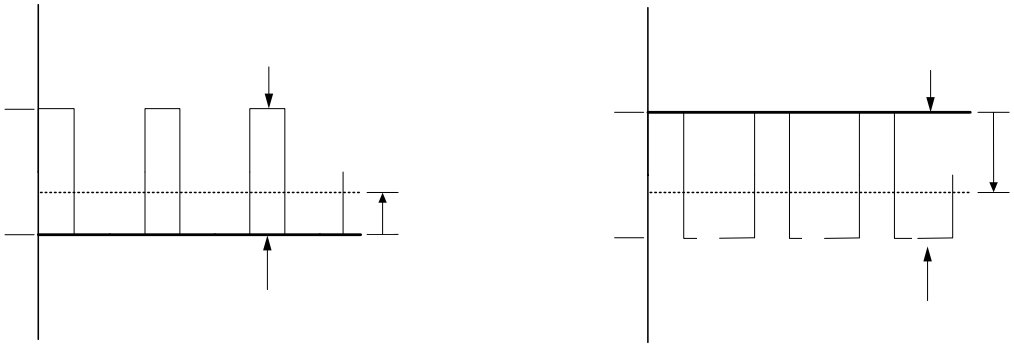


Figure 2.13: PWM Outputs of KoreMotor Board

2.6.2 KoreMotor Inter-board Communication

Each motor controller (microcontroller) has a primary and secondary physical address on the I2C bus and can be addressed directly by the KoreBot board. A set of dip switches shown in Figure 2.12 (b) is responsible for setting the mode of operation, as well as choosing the address range of the controllers (which is important when there is more than one KoreMotor board connected).

The operation of the I2C bus is relatively simple, nevertheless it serves as a vital communications path and an explanation will follow. Like serial communication, the I2C only requires two wires. Referring to Figure 2.14 (a) there is a line for data, SDA, and a line for the clock, SCL. Both lines have "open drain" drivers and can only be driven low, but pull-up resistors, labeled Rp, are used to pull the bus lines high. There are only master and slave devices on the bus. For example the KoreBot is the master and each PIC microcontroller on the KoreMotor board is a slave. The master is solely responsible for operating the clock line as well as initiating all data transfers, however, both devices are capable of driving the SDA line.

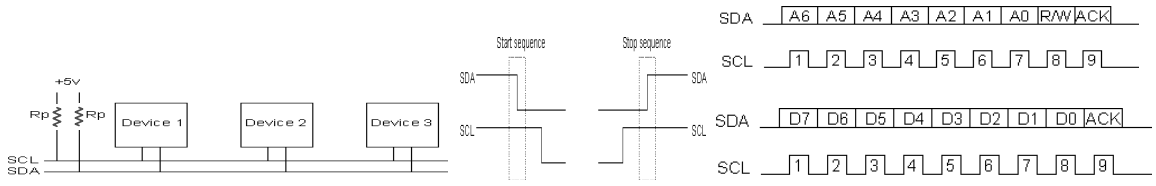


Figure 2.14: I2C Device Connections and Communication Protocols

When the KoreBot wishes to communicate with a motor controller it begins by issuing a start sequence and finishes with a stop sequence, see Figure 2.14 (b). After a start sequence the KoreBot sends the 8 bit address of a motor controller (0x0B – 0x0E), and the motor controller responds with an acknowledge bit (Figure 2.14(c)). However, when the Korebot then requests to read from or write to the contents of a register, the microcontroller must then go to an interrupt routine, save its working registers, and service the request, sometimes taking over a microsecond. During the interrupt routine the microcontroller is actually holding the SCL clock line low, and releases it when it is ready to complete the data transfer. Therefore, while the KoreBot boasts of having a fast

400 kHz I2C clock, it fails to make mention of the clock cycles wasted waiting on the microcontroller. This problem plagued the supervisory control software of the KoreBot and will be discussed in the next chapter.

2.7 Motor Controller and Amplifier Low-Pass Interface Circuitry

As mentioned before, interfacing the KoreMotor with the MSK 4360 amplifier posed a challenge. The current command to the amplifiers must be an analog DC voltage, yet the KoreMotor only outputs a digital PWM voltage. The straightforward solution was to build an analog low-pass filter to extract the DC component of the PWM signal. With the exception of the gain, the low-pass filters are identical for both the thigh and knee interface circuits.

To this end, Matlab was used to model the effectiveness of various filters at attenuating the 10 kHz component, and subsequent harmonics, from the PWM signals produced by the KoreMotor board. As expected, Figure 2.15 shows the dramatic amount of noise reduction between a first and second ordered filter.

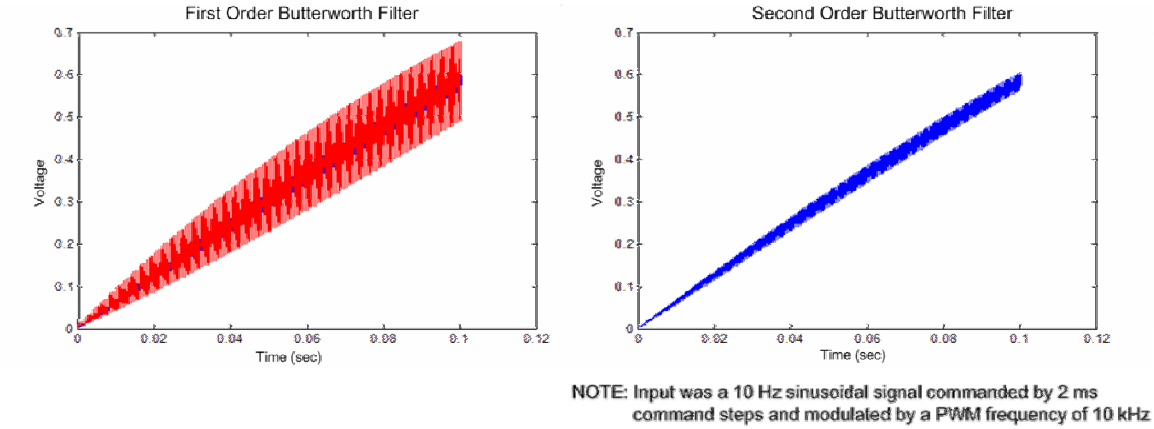


Figure 2.15: Outputs of First and Second Order Butterworth Filters

Further Matlab exploration with the filtered signal passing through the amplifier and motor models showed that a second order filter was much more desirable than a first order filter. First, the high frequencies not filtered by the first order filter could run the risk of creating instability in the current control loop of the MSK motor amplifier. Second, by choosing a higher-ordered filter the cutoff frequency could be increased, effectively increasing the bandwidth, and thus not interfering with the closed loop motor control performed on the KoreMotor board. Finally, the UAF42 integrated circuit (IC) by Burr-Brown was an easy-to-come-by second order filter and thus presented itself as the most viable solution.

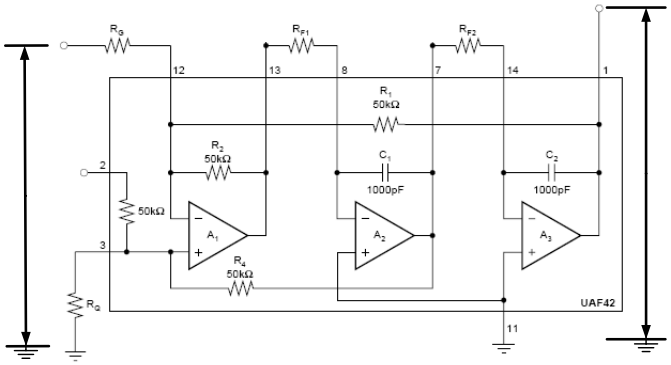


Figure 2.16: Burr-Brown UAF42 Single Input / Single Output Universal Filter

The 14-pin DIP package of the UAF42 in Figure 2.16 can be used to implement low-pass, high-pass, band-pass, and band-reject filters using either a Butterworth, Chebyshev, or Bessel filter type. Circuit analysis reveals that the transfer function between input voltage, V_{IN}, and output voltage, V_o, is

$$\frac{V_o(s)}{V_{IN}(s)} = \frac{A_{LP}\omega_n^{\,2}}{s^2 + s\dfrac{\omega_n}{707.1x10^6} + \omega_n^{\,2}} \quad \text{where} \quad \omega_n^{\,2} = \frac{R_2}{R_1 R_{F1} R_{F2} C_1 C_2} \quad \text{and} \quad A_{LP} = \frac{R_1}{R_G}.$$

First, the cutoff frequency or "3dB down" frequency should be determined. The control steps of the closed loop PID controller on the KoreMotor board occur every two milliseconds (500Hz), meaning that the circuit should not attenuate at or below this frequency. The commands are not sinusoidal so aliasing was not an issue and both circuits were assigned a cutoff frequency of, ω_n at 500Hz. The attenuation of the 10kHz PWM frequency component was calculated to be -51.91dB. The equation for ω_n^2 was then satisfied by using discrete off-the-shelf resistor values.

Second, the filters pass band gain must be found for each axis of control. Ultimately, the gain was determined so that at 100% PWM duty cycle the motors were commanded to be at a desired maximum current level. Recall that for every volt on ICommand, two amps are commanded to the motor and that the PWM voltage has a peak to peak value of 5 volts. Table 2.3 summarizes the gains that were selected for the two filters.

Table 2.3: Selection of Filter Gains

Axis	**V_{IN}** peak-to-peak (volts)	Maximum Desired Motor Current (amps)	Maximum Desired **ICommand** and **V_o** (volts)	Required Gain $\left(\frac{V_O}{V_{IN}}\right)$	Resistor **R_G** used (kohms)	Actual Gain, **A_{LP}** $\left(\frac{R_1}{R_G}\right)$
Thigh	5	3.6	1.8	0.36	150	0.33
Knee	5	10	5	1	47	1.06

Note, because standardized off-the-shelf resistors were used, it was not possible to exactly achieve the desired gain. However, the final circuits for each axis have a potentiometer in series with resistor R_G, and adjusted so that the desired gain was

obtained. See Appendix A7 for a complete list of resistors and wiring connections for the UAF42.

A drawback to using the UAF42 was that it did not offer differential input. As seen in Figure 2.16, V_{IN} and V_o are both referenced to a ground potential. A problem would arise in certain situations if the PWM output of the KoreMotor board were connected directly to V_{IN}. Recall from Figure 2.13 that the output of the KoreMotor board has a wire that determines direction as well as the wire that carries the PWM signal. When commanding a "positive direction" the direction line and the ground reference of the filter are both at the same ground potential, GND2. Therefore it would be acceptable to connect the PWM line and direction line of the KoreMotor to V_{IN} and GND2 on the UAF42, respectively. However, if they are connected when a "negative direction" is commanded, the direction line would be pulled to +5V while still being connected to ground on the UAF42, thus creating a short circuit. To remedy this problem a Burr-Brown INA114 general purpose instrumentation amplifier was front ended to each UAF42 filter.

The instrumentation amplifier serves to extract the effective differential voltage of the KoreMotor outputs shown in Figure 2.13, and establish a single bi-polar voltage referenced to ground. The instrumentation amplifier's setup is identical for both the thigh and knee circuits, as outlined in Appendix A7. Nevertheless, a short discussion will highlight some of the most important aspects of the amplifier.

The INA114 requires input circuitry to operate correctly. First, because the KoreMotor's outputs are meant to drive a motor, a resistive load must be connected across these lines. A simple resistor, of a few hundred ohms will provide enough current

to properly measure the voltage across these two lines. Second, another pair of resistors are needed, one from each of the input lines to ground, GND2 (Figure 2.17). These resistors provide a return path to drain off any input bias current. Without a bias current return path, the inputs will float to a potential which exceeds the common-mode range of the INA114, causing the input amplifiers to saturate. Third, the rails of the amplifier clip the output voltage at about 1.5 volts below the supplied voltage. Therefore, to prevent clipping and maintain transparency between the filter and the KoreMotor, the amplifiers were provided with a +/- 12 volt supply.

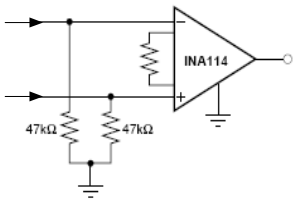


Figure 2.17: Input Circuitry for Burr-Brown INA114, Interface to Low-Pass Filter

2.8 Power Supply and DC to DC Converters

A Kepco RKE 900W switching power supply was chosen to provide the 48VDC bus. Two Texas Instruments DC to DC converters have a combined weight of only 33 grams and are used to convert the 48VDC into lower voltages for the embedded system and interface electronics. The 6 Watt PT4313 provides +/- 12VDC and the 20 Watt PT4520 provides +5VDC. As Figure 2.2 illustrates, the main 48 volt Kepco supply is not mounted on the moving system like the DC to DC converters. This setup allows the electronics package to remain lightweight and also reduces the amount of wires that must be tethered to the moving leg.

The Kepco RKE power supply is connected to an alternating current (AC) power source of 110 Volts, but can handle anything between 85VAC and 265VAC. The

48VDC output is highly regulated by pulse width modulation, and allows for the regulation circuitry on the MSK motor amplifiers to be kept to a minimum. Overvoltage protection and an isolated remote TTL on-off control is also provided.

Both DC to DC converters have isolated outputs with overvoltage protection. The 5VDC module can source 4 amps and is used to power the K-Team boards, wireless card, and supply voltage for the KoreMotor board's separate PWM supply. An external resistor was needed to adjust the module's regulated voltage to be just above +5VDC. The +/- 12VDC module can source half an amp on each output, and supplies the interface electronics (two INA114s and two UAF42s) along with the two CPU fans that cool the heat sinks. If the CPU fans are not connected, then there is not enough load on the module for it to operate correctly and 1kΩ resistors should be used to bypass each output to ground. The modules were sized so that they would meet the maximum loading conditions but remain as small as possible. Further specifications for the power supplies can be found in Appendix A8.

2.8 Summary

This chapter has presented an overview of the hardware integration for the robotic leg. The various components of the system were discussed, including the amplifiers, motors, optical encoders, interface circuitry, and power supplies. In addition, the main control connections for each axis were summarized. Finally, special emphasis was placed on explaining the characteristics of the motor drive system with an in-depth look at the internal architecture of the MSK 4360.

CHAPTER 3

SOFTWARE DEVELOPMENT FOR REAL-TIME COMPUTER CONTROL

3.1 Introduction

This chapter will discuss the assortment of software and development tools necessary to obtain real-time computer control of the robotic leg. In particular, special attention will be paid to the functions and commands that are used to control both the thigh and knee axes. The importance and difficulty of maintaining real-time computer control cannot be underestimated. In order to meet the timing deadlines required during real-time control the software must account for both execution and I/O time.

Two levels of control exist, supervisory and localized. The localized control is performed by a microcontroller (μC) on the KoreMotor board. Each μC provides an open loop interface as well as a closed loop algorithmic controller with several complementary features. The KoreMotor board can exist as a stand-alone module and receive commands from a serial RS232 link, however, to be used for demanding robotic applications it should be combined with the KoreBot supervisory and control board. The KoreBot embedded system is running a standard Linux distribution and provides all necessary tools for software development and execution. Applications can be created to run on the KoreBot system and make use of a powerful Application Programming Interface (API) to communicate with each μC on the KoreMotor board. This setup can meet a real-time supervisory control step time of 10 ms while at the same time maintain simultaneous closed loop control steps for each motor at 2 ms intervals.

3.2 Control

The goal of the control system is to provide accurate real-time control of the robotic leg shown in Figure 2.1 of the previous chapter. The two DC motors at the hip and knee provide the necessary actuation while the various encoders give adequate feedback. Figure 2.1 also provides insight into another control issue, that of coupling. The drive cable for the shank wraps around an idler on the hip axis before reaching the torsional spring at the knee. If the leg were free to move, a change in thigh deflection referenced to the z axis, called $\Delta\theta_t$, would cause the cable to wrap around the idler. As a result, the relative angle of the thigh and shank, θ_k, are displaced, but the angle of the shank relative to the z axis, θ_s, remains unchanged. By far the simplest way to understand how to control the coupled system, is to realize that the thigh motor controls θ_t, and the knee motor controls, θ_s. On the other hand, it should not be forgotten that when the leg is in contact with the ground, part of the work done by the knee motor may actually go into displacing the knee spring, θ_{ds}.

3.2.1 Control Architecture

Understanding the control architecture is fundamental in designing proper control software. Figure 2.1 clearly shows that the KoreMotor board effectively closes the servo-loop between the motor drive and feedback sensors. It also illustrates that the KoreMotor board communicates with the KoreBot board. Figure 3.1 serves to distill the control of Figure 2.1 down into a simple diagram which is identical for both axes.

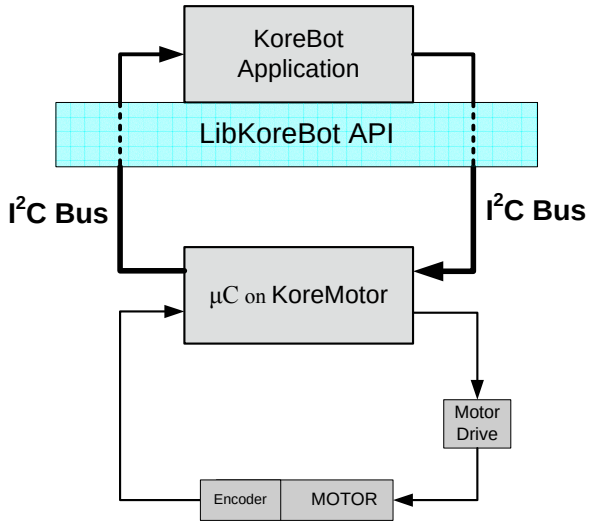


Figure 3.1: Control Architecture of a Single Axis

Referring to Figure 3.1, it should be clear that two loops of control exist. Each µC essentially closes an inner control loop with the Motor Drive and Feedback Encoders (bottom loop). At the same time, a supervisory control loop exists to command and request information from a microcontroller (top loop).

Each microcontroller on the KoreMotor board can control the motor in one of two ways. The most basic is open loop control, where by the controller's input is passed onto the motor by directly setting the output PWM duty cycle. Recall from Chapter 2 that the PWM signal is used to command motor current, therefore, open loop control can be used to directly set the motor current. The KoreMotor also offers closed loop control to regulate anything from position, speed, and acceleration, to a profile that combines the three. Each regulation will use a specific set of proportional, integral, and derivative (PID) coefficients, depending on the parameter which is actually being regulated. These coefficients must be experimentally tuned once the system it is controlling is in place. The process, which is usually thermally and mechanically taxing on the system, will be discussed in the next chapter.

3.2.2 Real-time Control Considerations

The time it takes to close a control loop is usually referred to as a control step. Each control step typically requires calculating next state values based on present state information, commanding these values, and receiving feedback from the system. A key difference between the two control loops of Figure 3.1 is the rate at which they can issue control steps. The microcontrollers on the KoreMotor board are dedicated to closing the command loop in a desired and regular control step time. An application's process running on the KoreBot, however, must compete for processor and input output (I/O) resources to communicate with a single microcontroller. This introduces delay and does not allow the KoreBot to close the loop as fast as the motor controllers.

The combination of efficient coding techniques and the fast 400MHz CPU on the Linux system keep processor delays to a minimum. In addition, the Linux system is set up so that there are only a few other processes competing for CPU cycles. Any user created application gets priority scheduling of CPU time. Nevertheless, if more processes are added or process forks are created, careful consideration must be paid to the CPU scheduling.

The actual timing bottleneck occurs as a result of I/O calls. In order for an application to communicate with a motor controller it must use a serial communication bus called the inter-integrated circuit (I2C) bus. Only one such bus exists between the KoreMotor and KoreBot boards. As Section 2.6.2 pointed out, the KoreBot acquires master control of the bus and communicates with only one device at a time. In addition, Section 2.6.2 mentioned that the bus actually becomes locked by the motor controller until the entire data transaction is complete. The result is the time delay depicted in

Figure 3.2. The user's control process on the KoreBot requesting the I/O with the motor controller must be blocked until the motor controller has finished servicing the request. Without a real-time kernel running in Linux the operating system will schedule processor time for other kernel processes. Once the I/O is complete and the I2C bus is released by the μC, the CPU must perform a context switch, moving out the kernel process and returning the user's control process. The result is a net time delay proportional to how long the μC takes to service the I/O transaction and return control of the I2C bus. With a few exceptions, the time delay in the μC was experimentally determined to be directly related to the number of bytes each transaction contained.

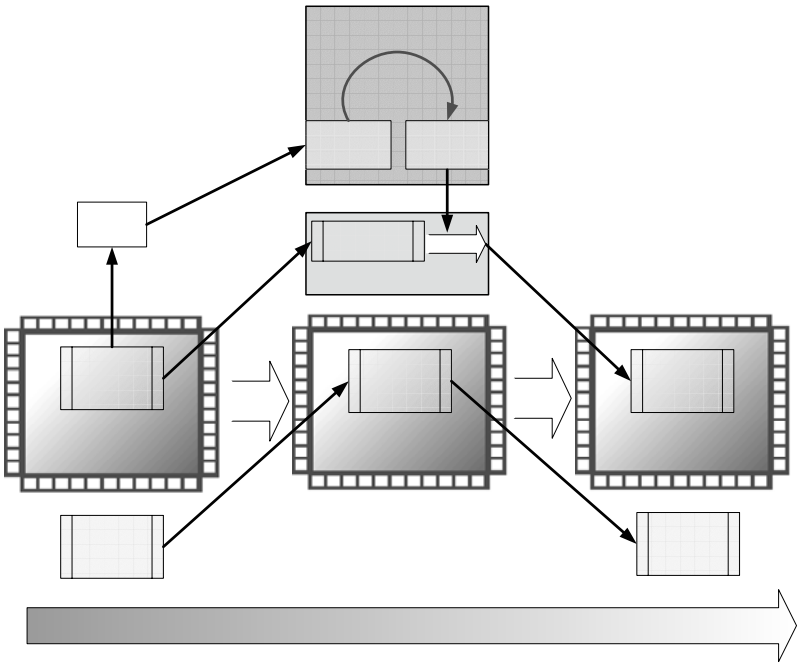


Figure 3.2: Timing Diagram of I/O Request

Setting the user process up for non-blocked I/O, or perhaps forking child processes, would not serve the intended purpose of speeding up the I/O. Forking off to a new process would not free up the I/O. Recall that the I2C bus is locked until the transaction is complete. Even complex computations and API calls take just a few

microseconds on the 400MHz CPU, while the I/O transaction can last a few milliseconds. In addition, the computations that need to be performed in the CPU are usually dependent upon the data obtained from the motor controller, so waiting becomes necessary.

In light of the timing delays in the control loop, Figure 3.1 should be refined. The overall architecture remains the same, however, a new figure will incorporate the time delays. Figure 3.3 shows four delay blocks: t_{RS}, t_{RP}, t_{IC}, and t_{CS}. The delays t_{RS} and t_{RP} stand for the time it takes to request speed and position data, respectively, and t_{IC} represents the time required before the issued command becomes an actual input command to the μC's control routine. The delay t_{CS} is simply the motor controller's control step time. When requests for data are serviced, they return the present state information that was stored just before the transaction was initiated. When commands are issued, the KoreMotor μC does not use the command as input until the next control step takes place, which can be as long as t_{CS} time later.

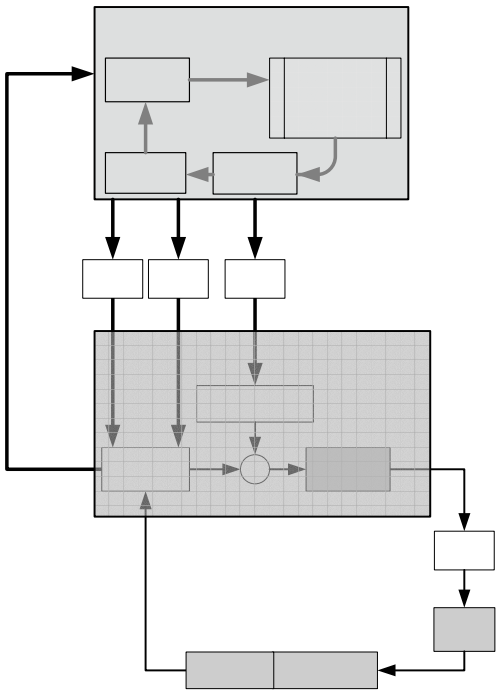


Figure 3.3: Detailed Control Architecture with Time Delays

It is very important to see that the present state data can be updated by the KoreMotor control loop between requests for speed and position data. That is, if a request to read the motor's speed is serviced near the end of a control step, then the present state information can be updated before the request for the position data is serviced. This delay between requests is directly related to the delay depicted in Figure 3.2. The user application must be brought back into scope before the next request can be executed. A similar scenario exists for the issuing of control commands. If a control command is issued but it takes t_{IC} time for the μC to get the command, the worst case would occur if that command arrived just after the motor controller began another control loop. A total of $t_{IC} + t_{CS}$ time delay would have accumulated before the controller actually uses the new command as input to its control routine. All this must be considered before taking on the task of developing supervisory control software to run on the KoreBot.

3.3 Software Development

This section will address how supervisory control software can be developed using the existing API and cross platform C compiler supplied by K-Team. It will also touch on some important functions and procedures that were created to assist in the development of control software for the robotic leg. A very basic understanding of the C programming language is necessary.

3.3.1 The Software Development Environment

In order to create and compile a C program that is executable on the KoreBot, the ARM-Linux-gcc tool chain and compiler must be properly installed. This was already documented in Chapter 1 of Adam Porr's "Hopper Project" entitled *Setting Up the KoreBot Development Computer*, a copy is included in Appendix A9. While his work also touched on how to install the necessary KoreBot libraries, K-Team has recently provided updated libraries and installation instructions that should be followed instead. For the most recent libraries go to: http://ftp.k-team.com/korebot/libkorebot/. Unzip the tar file into the home directory and follow the step-by-step instructions in the README file.

Included in the zip file are documented utilities and test programs. In particular the directory "`./libkorebot-1.x/template/`" should contain a C file called "`prog-template.c`" which already includes the necessary libraries for software development. Assuming the steps were followed correctly from Adam Porr's work, the file can be compiled into ARM processor machine code for executing on the KoreBot's ARM-Linux platform. As of version 1.7 of the LibKoreBot, programs compiled with dynamically linked libraries were not functioning correctly. The remedy is to use static linking, where all libraries are embedded into the program executable at compile time. This makes for a slightly larger executable, but ultimately loads faster because libraries do not have to be linked at execution time. Compiling the file statically is detailed by the accompanying "Makefile," or just type: `make template-static.`

3.3.2 KoreBot Application Programming Interface

K-Team has created an easy to use API, called LibKorebot, that provides high level C functions for interfacing with both the KoreMotor and KoreIO peripheral boards over the I2C bus. By doing a `#include <korebot/korebot.h>` the entire library is brought into the scope of a C program. A wide variety of commands are available, allowing the user to specify motion and controller configurations; as well as measure encoders, status, and error registers. A complete listing of the commands and their descriptions are given in Appendix A10. The library API is documented with Doxygend and is available online at: http://ftp.k-team.com/korebot/libkorebot-doc/files.html. For the purpose of this work, only the most relevant control commands are elaborated upon.

There are two steps to opening a connection with a motor controller. First, a handle, of type `knet_dev_t`, must be created for each module that will be used. For example, if there are plans to use two motor controllers on the KoreMotor board, then the following two lines are needed:

```
static knet_dev_t * hip_motor;
static knet_dev_t * knee_motor;
```

Another step is responsible for opening the communication path to a microcontroller by giving the handle access and management to the remote device. In order to use the open command, a string with the name of the μC must be used. The strings are not included in K-Team's documentation but can be found in Table 3.1.

Table 3.1: List of Literals Needed to Open Each Device

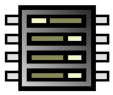
Device	Literal Name	I2C Address	Dip Switch Positions (Reset Button on the Right)
KoreMotor – Motor 0	"KoreMotor:PriMotor1"	0x0B	
KoreMotor – Motor 1	"KoreMotor:PriMotor2"	0x0C	
KoreMotor – Motor 2	"KoreMotor:PriMotor3"	0x0D	
KoreMotor – Motor 3	"KoreMotor:PriMotor4"	0x0E	
KoreMotor – Motor 0	"KoreMotor:AltMotor1"	0x0F	
KoreMotor – Motor 1	"KoreMotor:AltMotor2"	0x10	
KoreMotor – Motor 2	"KoreMotor:AltMotor3"	0x11	
KoreMotor – Motor 3	"KoreMotor:AltMotor4"	0x12	
KoreIO	"KoreIO:Board"	0x1C	
KoreIO	"KoreIO:AltBoard"	0x1D	

Figure 2.12 (a) can be used to find the physical location that the strings represent on the KoreMotor board. If the hip and knee motors are connected to motor ports 0 and 1, respectively, then the following lines of code will establish links with the correct motor controllers over the I2C bus:

```
hip_motor = knet_open( "KoreMotor:PriMotor1", KNET_BUS_I2C, 0, NULL );
knee_motor = knet_open( "KoreMotor:PriMotor2", KNET_BUS_I2C, 0, NULL );
```

If `knet_open` returns a zero then the controller did not open successfully. Figure 2.1 depicts how each motor controller is connected in this project. The same general convention is used to open additional motor controllers. Note, also, that at the end of a program, the `knet_close` command must be called to close communication with each

opened microcontroller. This command takes the handle of the controller as its only parameter.

It is now possible to exploit the high-level functionality of the LibKoreBot API. The device handles are used as the first argument in all high-level function and procedure calls. Before issuing commands to the hip or knee actuators, it is wise to run the initialization routine provided in Appendix A11. The routine will reset and reconfigure each controller. Perhaps the most important function of the initialization routine is the one that sets the controller's sampling period or control step time, t_{CS}. The sampling time is set in 1.6μS intervals. For example, to set the hip controller to use a sampling time of 1.6ms a value of 1000 is used as follows:

```
kmot_SetSampleTime ( hip_motor, 1000 );
```

To retrieve position or speed information from an encoder attached to a particular controller is simple. Two separate calls to the same function are made and differ only in their arguments. Recall that there is a time delay associated with each call. As a result, one function must wait until the other has returned from blocked I/O. The following function calls return long integer values:

```
positionVal = kmot_GetMeasure ( hip_motor, kMotMesPos );
speedVal = kmot_GetMeasure ( hip_motor, kMotMesSpeed );
```

The integer corresponding to position is in terms of encoder counts and can be converted to motor revolutions with the aid of Table 2.2. The integer corresponding to speed is the number of encoder counts per sample time. The motor speed can be determined using:

$$Motor\ Speed = \frac{speedVal}{SamplingPeriod \times Resolution}.$$

Issuing a control command to a motor is very straightforward. The following procedure serves this purpose, and its arguments ultimately determine how the motor controller will behave:

```
kmot_SetPoint (knet_dev_t *dev, int regtype, long setPoint)
```

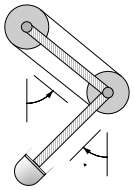
The control procedure has three parameters. The first, as already mentioned, is the handle for the desired motor controller. The second parameter determines how the controller will regulate the motor command. The third is the input command value for the controller (Figure 3.3), and depending on the regulation type determines the motor's position, speed, or current. LibKoreBot has defined enumerations for the various modes of regulation. During coding it can be useful to replace the lengthy enumeration with its corresponding integer provided from Table 3.2.

Table 3.2: Enumeration Table for Regulation Types

Enumeration	Integer	Description
`kMotRegOpenLoop`	0	Open Loop Current Command
`kMotRegPos`	1	Closed Loop Regulation on Position
`kMotRegPosProfile`	2	Closed Loop Regulation on Position with a Speed Profile
`kMotRegSpeed`	3	Closed Loop Regulation on Speed
`kMotRegSpeedProfile`	4	Closed Loop Regulation on Speed with an Acceleration Profile

The sign of the command value in the third argument is related to the direction of the motors. Table 3.3 can be used to quickly reference the direction of each axes in terms of the parameters from Figure 2.1.

Table 3.3: KoreMotor Directions

Axis	Positive Command Positive Encoder	
Thigh	$-\theta_t$	
Knee	θ_s	

Special attention is needed when issuing open loop current commands. The PWM duty cycle has a resolution of 9-bits in each direction, meaning that open loop commands can vary from +512 to -512, with the sign indicating the direction. K-Team incorrectly documents 10-bits of resolution in each direction. Furthermore, a flaw in the firmware of each μC spreads the 9-bits of resolution across 15-bits. That is, it appears to have an adjustable duty cycle that can vary from +32767 to -32767, but only a total of

512 of the values actually affect the duty cycle. Unfortunately, the 512 values are not arranged linearly across the 15-bit range. While the updated Firmware on a new KoreMotor board put the command range correctly between +512 and -512, it failed to fix the issue of non-linearity. The first KoreMotor board was kept, and trial and error was used to find the desired open loop commands within the 15-bit range. Sample open loop commands that output zero, half, and full duty cycle are:

```
kmot_SetPoint( knee_motor, kMotRegOpenLoop, 0)      //zero duty cycle
kmot_SetPoint( knee_motor, kMotRegOpenLoop, 20000) //half duty cycle
kmot_SetPoint( knee_motor, kMotRegOpenLoop, 32767) //full duty cycle
```

Before issuing a closed loop command, a functioning set of PID coefficients for the desired regulation type must be loaded into the controller. The procedure provided by the library for setting the coefficients has five parameters. The first, as usual, is the controller's handle. The second is the desired mode of closed loop regulation. The last three arguments are the 16-bit integer values for the proportional (kp), derivative (kd), and integral (ki) coefficients. For example, to set the PID coefficients for closed loop regulation on position the code would appear:

```
kmot_ConfigurePID( knee_motor, kMotRegPos, kp, kd, ki );
```

Be sure to pay careful attention to the counter-intuitive ordering of the coefficients in the last three arguments. Also, the coefficients will only be used for closed loop regulation on position and new gains have to be configured for the other

regulation types before they can be used. Now the closed loop regulation command for position can be used. In another example, the knee motor will be moved -6000 encoder counts, or an eighth of a revolution, in the $-\theta_s$ direction as follows:

```
positionVal = kmot_GetMeasure ( knee_motor, kMotMesPos );
kmot_SetPoint( knee_motor, kMotRegPos, positionVal - 6000);
```

It is also useful to be able to calibrate the system by resetting the encoders to zero, or to some other known value. The `kmot_SetPosition` command is used to set an encoder to a particular value, where the first argument is the motor controller's handle, and the second is the new encoder value.

The last of the API utilities that need to be discussed involves sensing the digital input of the KoreIO board. A handle is created the same way it was for the motor controllers, and the appropriate literal is used from Table 3.1 to open the communication with the KoreIO board. The function is quite simple. Like the previous functions, the first argument is the handle for the KoreIO board. The second argument is an integer signifying which of the 16 digital inputs is to be read. The resulting digital value ('0' or '1') is returned as an integer. With the embedded micro switch foot sensor of Figure 2.1 connected between digital input port one and +5 volts on the KoreIO board, the value for σ can be found as follows:

```
sigma = kio_ReadIO(koreio,1);
```

As mentioned in the discussion of real-time processing, there is a delay associated with each API function and procedure. The delay is a result of having to perform I/O calls to the peripheral boards through the I2C bus. Each function and procedure differs in

the demands it places on the I2C bus. Some require only the transmission of a few bytes, others require bytes of data to be both transmitted and received. With the need to meet real-time constraints, it becomes extremely useful to know how long each function and procedure takes to complete. For this reason, Table 3.4 was compiled and provides a list of the average time delays experienced by calling each function at least 2000 times.

Table 3.4: Average Time Delays for High Level API Calls

Function and Procedure Call	Time Delay (ms)
`kmot_GetMeasure - position` (t_{RP})	1.80
`kmot_GetMeasure - speed` (t_{RS})	0.90
`kmot_SetPoint` (t_{IC})	2.37
`kmot_SetSampleTime`	0.64
`kmot_SetPosition`	1.28
`kmot_ConfigurePID`	1.90
`kmot_SetSpeedProfile`	0.96
`kio_ReadIO`	0.47

It is known, however, that each command transaction over the I2C is handled entirely by the Intel processor. All the transmitted bytes are first buffered at the hardware level, and an integrated I2C unit handles the entire transaction with the peripheral device. Nonetheless, the average delays are not absolute. Depending on how long the microcontroller's on the peripheral boards take to respond and how long Linux takes to switch back in the blocked process, the total time was seen to vary significantly about the average. Thus, it would be extremely unadvisable to attempt to close the motor servo loop through a KoreBot application.

3.4 Summary

This chapter discussed both the control aspects and software development of the KoreBot and KoreMotor boards. In particular, a general overview of the control architecture was given, with a more detailed look at how real-time demands could be met. Then, a look at the software development outlined what was needed to write and compile an application for the KoreBot Linux platform. Finally, a discussion of the LibKoreBot C API was presented, including a discussion of several important supervisory and control commands. These will become important in the next chapter, when the details of the control code for the jumping leg are presented.

CHAPTER 4

RESULTS

4.1 Introduction

The successful integration of high level control software with the embedded system, motor drive, and interface electronics made it possible to run a series of experiments on the robotic leg. The tests showed promising results, in particular, the ability to achieve a high performance jump. Varying levels of success were achieved with subsequent executions of the jump routine on the actual leg. It was found that changing the initial leg positions significantly altered the overall jumping performance. Nevertheless, a set of initial conditions were found that created an impressive jump that sent the moving plate to a vertical height of 50 cm. This chapter will discuss how the various components were used, and problems overcome, so that the objectives of this project were successfully completed.

4.2 Implementing the Jump in Software

A control application was developed to test the combined functionality of the hardware, software, and mechanical system. The application was set up so that the user could execute various routines by means of a command prompt interface. The routines ranged in complexity, from accessing basic API calls at runtime to executing a jump or kick maneuver with the leg. The code for the console application can be found in its

totality in Appendix A11. The most important of the routines created in the KoreBot application was the jump routine, informally called “hop” in the code.

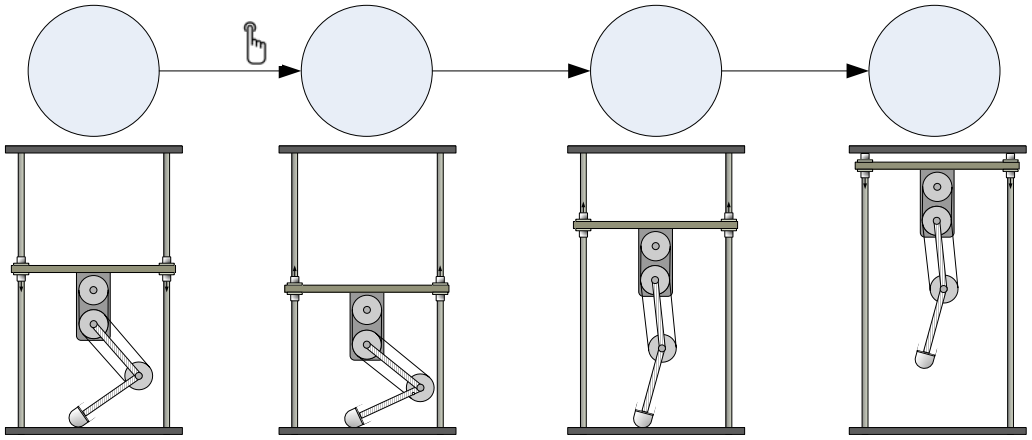
The jump routine was developed using the high level API functions and procedures outlined in the previous chapter. The routine made use of the KoreMotor’s open and closed loop regulation modes, specifically closed loop position control. Before closed loop control could be used, however, adequate PID coefficients had to be found. A procedure known as the Ziegler Nichols Method was followed and is outlined in Appendix A12. It is important to note that the Ziegler Nichols Method subjects the motor to repetitive impulsive oscillatory current commands. Quite often these currents have peak values at or near the maximums set up for each motor (10 amps for the knee actuator and 3.6 amps for the hip actuator). At these power levels the motor windings can quickly heat up to their maximum rated temperatures, and can easily become damaged. For this reason, the motors should be disabled in-between each test by using the disable switch connected to the two MSK amplifiers. It is also good practice to take frequent temperature readings and if necessary, allow the motors ample time to cool before proceeding.

Perhaps the most difficult aspect of implementing the jump routine was correctly sensing ground contact, while maintaining real-time supervisory control. Due to the time delays associated with each API call, care had to be taken in both debouncing the contact sensor and obtaining encoder data at regular intervals. Often times simple vibrations can cause the contact sensor to jitter and falsely detect contact. To overcome this problem, a method of debouncing the contact sensor was employed. The debouncer serves to filter the transients, or jitter, from the micro switch to determine the true state of the foot

sensor. To do this, consecutive samples of the digital input must be taken. Experience has shown that at least 3 ms is sufficient time for allowing the transients of a contact to diminish.

In a typical control sequence, the leg's state must be updated by pulling information from all four encoders and the contact sensor. Table 3.4 from the previous chapter shows that it takes, on average, 1.8 ms to obtain the position of each encoder and 0.47 ms to check the contact sensor. In order for the application to update the state of the entire machine, a total of 7.7 ms would have to elapse. An additional 2.5 ms is needed to effectively debounce the contact sensor giving a total refresh rate of 10.2 ms. In a worst case analysis, if contact were to occur just after polling the digital IO, then the system would not detect and debounce this until nearly 18 ms later. An additional 2.37 ms is needed to command a state change to each motor, effectively increasing the worst case scenario to 22.7 ms. Clearly, this is dependent on how much encoder information is required. If experiments can be focused to study the information from fewer encoders, then the control step time can be reduced dramatically.

The jump routine served the purpose of controlling the leg through the four phases of a primitive jump. As Figure 4.1 illustrates, the four phases can be combined to form a state machine.



user

Figure 4.1: State Machine and Leg Phases for a Single Jump

Stance

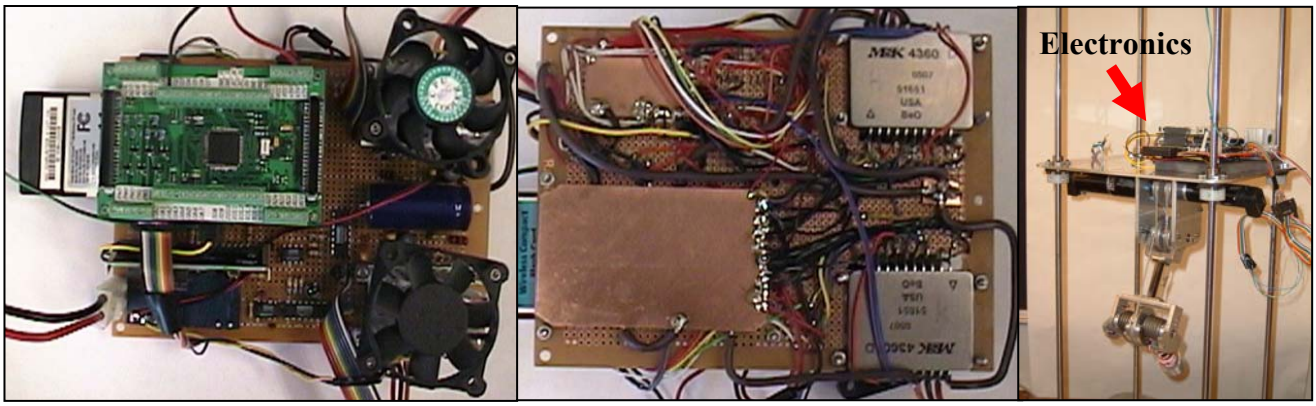
In the first state, the application commands each motor to maintain its joint angles by means of closed loop PID regulation on position. The leg effectively remains in a stance position until the user initiates the jump sequence by pressing any key. The key press transitions the application from the Stance state into the next state called Crouch. In this state, the actuators are in a sense "turned off" so that the leg can fall into a crouch. Because the knee has a larger gear ratio, the back drive characteristics impeded compression at the knee joint. In order to effectively "get the knee out of the way," a small open loop current is commanded at this joint to help retract the knee. It was experimentally determined that after 50 ms in the Crouch state, the leg is almost maximally compressed. A high resolution software timer detects the elapsed time and transitions the leg into the Thrust state. During the Thrust state, the motor controllers are in open loop mode, commanding maximum current to each motor. Once the application has determined that the foot has broken contact with the ground ($\sigma = 0$), the controller is transitioned into its final Flight / Touchdown state. This state serves a dual purpose by providing both flight and touchdown control. The same set of PID gains used in the Stance phase are enacted to provide closed loop regulation and reposition the leg for

$\sigma=1$

touchdown. The application remains in this state while the leg makes contact with the ground, compresses into the stance position, and becomes quiescent.

4.3 Results and Discussion

The embedded system, motor drive, and interface electronics were successfully integrated onto a 6.25" by 4.5" perforated board and mounted onto the moving system as seen in Figure 4.2. In keeping with the low-weight restriction, the entire electronics package weighed in at just under 500 grams. Despite the less than appealing aesthetic quality of the wiring, the electronics are very durable and have been able to withstand numerous high impact jump tests.



(a) (b) (c)

Figure 4.2: (a) Top and (b) Bottom View of (c) Electronics Mounted to Robotic Leg

As mentioned, the power stroke that thrusts the leg into a jump depended on, among other factors, the initial leg position. The two key components were the foot position and leg compression. Systematic trial and error was used to determine a leg configuration that resulted in the longest power stroke and highest vertical jump. At first, the foot was positioned directly under the hip axis, and the time spent in the Crouch state

was altered to allow for different degrees of leg compression. In subsequent trials, the foot position was gradually moved towards the negative x direction, and various leg compressions were tested. It was concluded that when the foot was displaced -1.19 inches from the hip axis as depicted in Figure 4.3, it resulted in the longest power stroke and best vertical displacement.

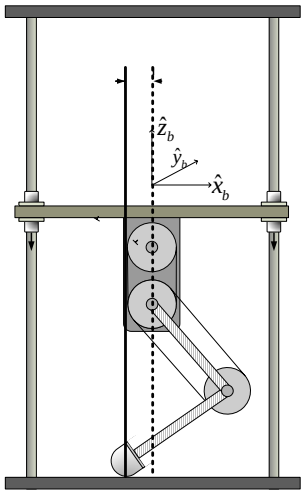


Figure 4.3: Initial Foot Position for Best Vertical Jump

As mentioned earlier, the top of the moving plate reached a maximum height, *h*, of 50 cm. However, this number can be deceiving as the hip axis actually resides a distance l_h below the top of the moving plate. Furthermore, simulation by Darren Krasny, measured height as the distance from the ground to the hip axis. Therefore, it is only proper to adjust the height by l_h before comparing the actual performance with the theoretical.

A plot was produced of the height of the hip axis during a jump. Of course this was not measured directly, rather the string encoder measured the distance between the top plate and the moving plate, h'. After the data was converted into the correct height, it was plotted against the expected height from Krasny's simulation (see Figure 4.4.)

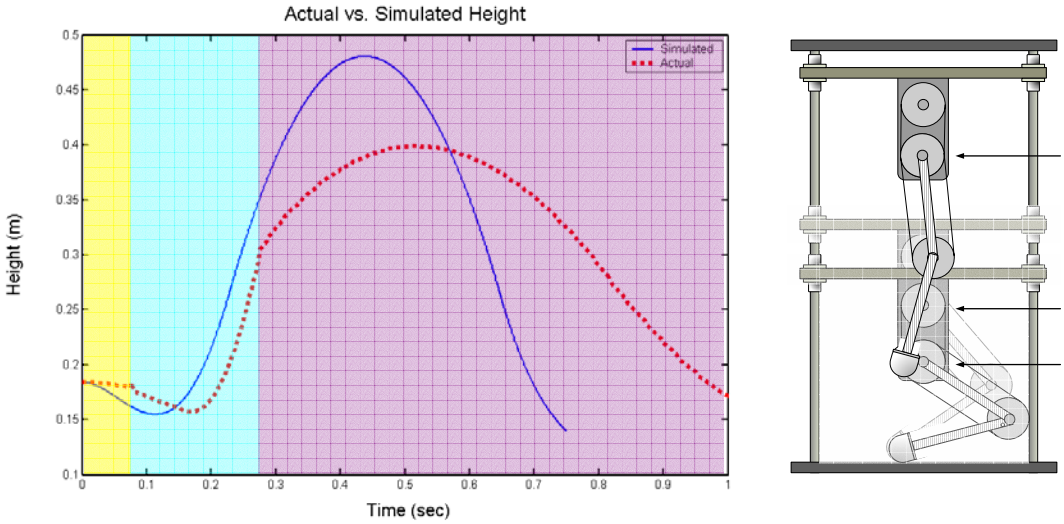


Figure 4.4: Simulated vs Actual Height Measured at Leg's Hip Axis

From the plot we can identify the four leg phases of the jump. During the Stance phase, the leg remained locked with the hip axis 18.5 centimeters from the ground. At time zero the leg is transitioning out of a stance and into a crouch. Because of the Crouch phase the leg compressed another 4 centimeters until the hip was only 15.5 cm from the ground. The transition to the Thrust phase, however, occurs before the leg becomes maximally compressed. That is, the sudden application of the torque cannot create an instantaneous change in the leg's vertical velocity. Recall that the motor torque must be transmitted across the torsional spring before the shank can convert it to vertical thrust, adding further delay. The software recorded that the leg remained in the Thrust state for 204 milliseconds. This is confirmed by the height data, which shows a change in vertical velocity 200 milliseconds after the beginning the thrust phase and corresponding to a height of 30 centimeters, roughly the height of the fully extended leg. The leg remains airborne for a considerable amount of time, and the hip axis reaches a maximum height of 40 cm. In all, the hip sees nearly 25 cm of vertical displacement from the maximum

compression to the top of the flight. A frame by frame capture of an actual jumping sequence can be seen in Figure 4.5.

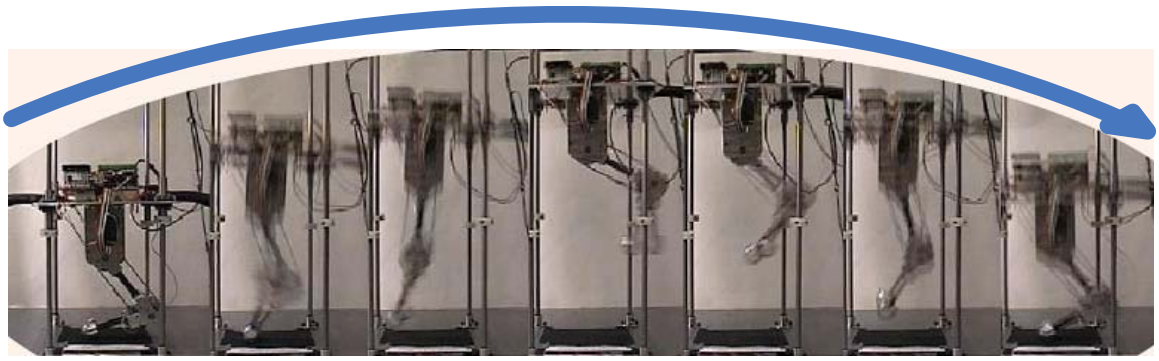


Figure 4.5: Frame-by-frame Capture of Actual Jump

Figure 4.4 leaves more to be discussed about the overall performance of the leg. An obvious feature is the inability for the leg to reach the simulated height of 47 cm. While it is only a simulation, it should be accredited with having modeled nearly every dominate characteristic of the system, from motor efficiency and back drive properties, to rail friction and series elastic actuation. Possible shortcomings and isolated failures of the mechanical system could contribute to the degradation in performance. Upon disassembly of the leg, the gear boxes of both actuators showed signs of having damaged bearings. In addition, the shaft of the hip motor had actually been bent. Tests on the hip actuator alone showed deteriorated performance from earlier no-load tests. It is quite possible that these factors existed throughout the jump tests and could have contributed, in a large part, to the reduced vertical jumping height shown in Figure 4.4. The damage was determined to be a combination of mechanical design and assembly issues that will be resolved in time.

4.4 Summary

This chapter has presented the most pertinent results of this project. Specifically, there was emphasis on the ability of the leg to achieve a high performance jump. One such jump was recorded as having sent the moving plate 50 cm into the air. Special attention was also paid to the state based software responsible for producing the jump. There was an important discussion on how to tune the PID controllers and prevent overheating of the motors. At the same time, a look at the supervisory control step during a jump revealed some of the problems with having time delays associated with each command. A broad discussion of the final electronics package was also given.

CHAPTER 5

SUMMARY AND CONCLUSIONS

5.1 Summary and Conclusions

Aside from the hardware issues mentioned in the previous chapter, the results of testing the jumping leg indicated that the research goals of this project were successfully completed. To be sure, two-axis coordinated control of the jumping leg was clearly established and demonstrated through repeated successful high performance jump tests. Video archives of these sessions are available from Dr. David Orin of the Department of Electrical and Computer Engineering at The Ohio State University.

The ability to successfully integrate the eclectic mixture of hardware components with each other and functioning towards the desired end with the software was a challenge that cannot be underestimated. In the end, the amplifiers and their supporting electronics were capable of driving the motors without problems. Despite the issue of heating, the motors and their accompanying gear boxes showed satisfactory performance in actuating both axes. Finally, it should be reminded that the embedded system, motor controllers, and digital IO boards were all fairly new products by K-Team, and for the most part, lacked adequate documentation. Nevertheless, the host of software and hardware issues were sorted out, and the overall system's performance was acceptable.

In general, this research has proven to be a significant step towards studying the characteristics of a series compliant articulated jumping leg. It is expected that future research will use this work to perform an in-depth analysis of the legged machine and to this end, aid in the development of a truly dynamic galloping quadrupedal machine.

5.2 Future Work

The next tasks that must be completed for this project involve finding a solution for the mechanical system so that it does not become damaged during assembly. Luckily, design alterations are underway that seek to replace the brut force assembly method with a new tensioning system. In parallel to that, work is also being conducted to create a more realistic model of the leg in simulation using the very robust graphical simulator known as Robot Builder. The eventual completion of this simulation will make it possible to better understand and further explore the properties of the series compliant leg, that was not easily understood by the Matlab simulation produced by Darren Krasny. In addition, an effort should be made to better model the physical system in simulation by conducting a thorough set of system identification tests with the leg. Once completed, the simulation could also be of great use in determining a new motor and gear box for each axis, both key properties that have yet to have been optimized.

Another objective that will no doubt make this project novel, is to incorporate an evolutionary search algorithm into the actual hardware. Work in this area could explore two new and exciting topics. First, optimization could be performed entirely in hardware using repetitive evolutionary search techniques. Eventually, it could be used to find everything from complex current profiles, to initial leg configurations that enable the leg to have an even higher degree of performance than was observed in this project. Second, the evolutionary algorithm could be used to optimize parameters in simulation which are then ported to the actual hardware for testing. A comparison of the actual results with those evolved in simulation could help to reveal the importance of performing evolutionary based searches in simulation.

A very important, yet relatively simple test should precede any such testing. Due to the repetitive nature of evolutionary tests, it must be confirmed that the mechanical and real-time systems are fit for the task. The results of which will also have direct implications on the feasibility of replicating the leg in order to produce a quadruped capable of dynamic galloping. A control application should, therefore, be written for the KoreBot that creates a state machine similar to the one presented in the last chapter and employed in this project. An additional state and transition would be needed to sense ground contact and bring the machine back to the crouch state, thus allowing for continuous jumping to occur. Depending on the results, it may become clear that the current distribution of Linux running on the KoreBot is not sufficient for the task and the operating system may need to be upgraded to include a real-time kernel.

Finally, if the leg is to be installed onto a future galloping machine, there is one further scenario in which the leg's performance should be evaluated. It would be extremely useful to secure the leg in the air, having it cycle through a somewhat circular motion, whereby the foot would be seen from the side as tracing an oval through the air. The speed at which the leg cycles through the air will have a direct bearing on the maximum attainable speed of the future quadrupedal robot.

Clearly there is still productive and novel research that can grow from this project. Everything from continuous steady state jumping, to simulation and evolutionary optimization should be addressed. Given that this project has set the stage for exciting new research, hopefully future work will lead to the development of a dynamically stable galloping quadruped.

Appendix A1: Values for Physical Parameters of Leg

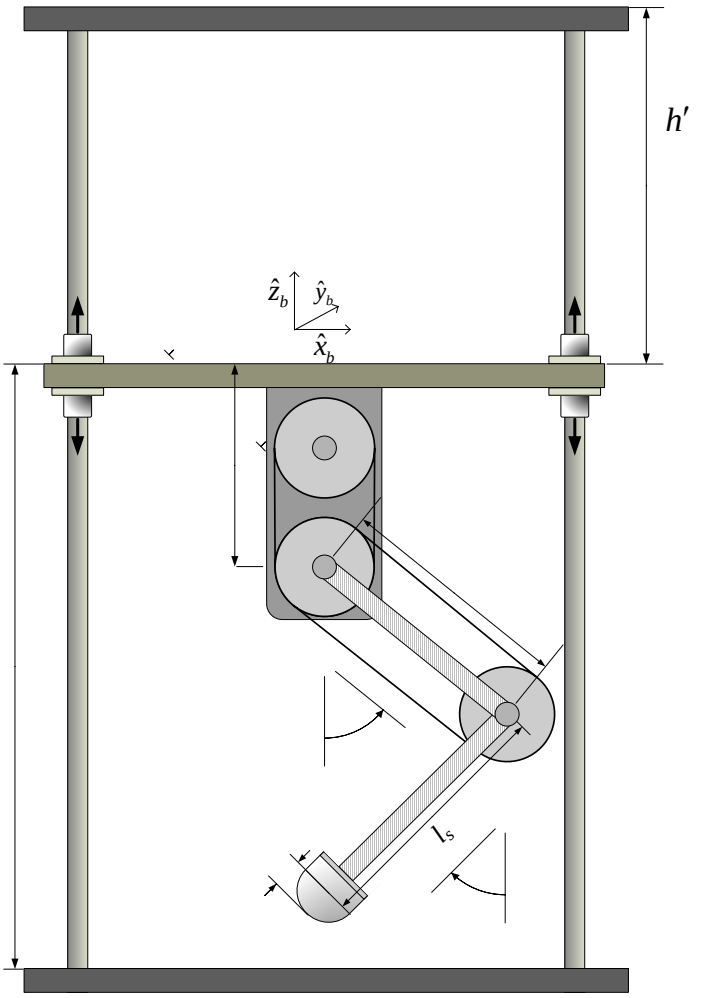


Figure A1.1: Physical Representation of Leg

Table A1.1: Physical Leg Parameters

Parameter	Length (cm)
l_t	14.0
l_s	12.7
l_h	10.8
r_f	1.90

APPENDIX A2: Electrical System Diagram for Embedded System, Motor Drive, and Interface

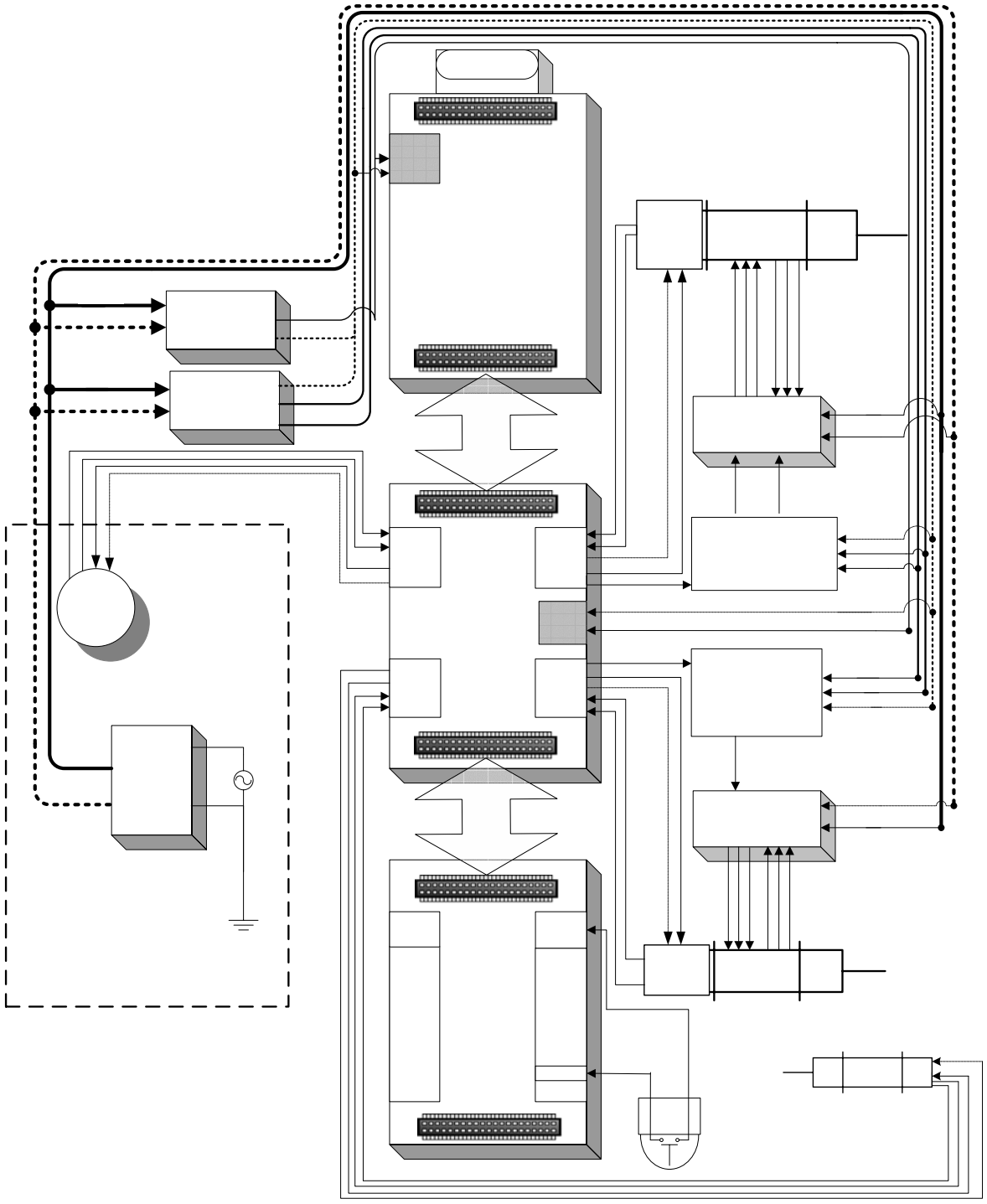


Figure A2.1: Electrical System Diagram

APPENDIX A3: MSK 4360 Motor Amplifier Connections and Specifications

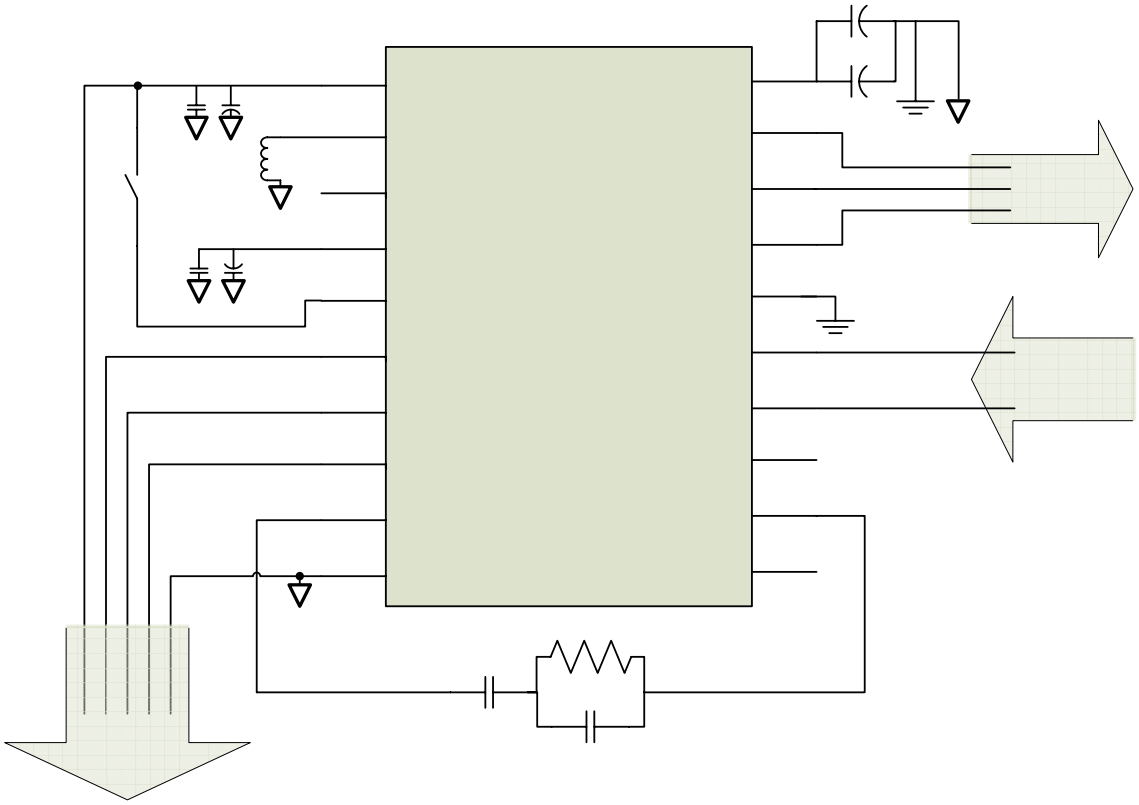


Figure A3.1: Wiring Diagram for MSK 4360 Motor Amplifier

Specification	*Range*
DC Voltage Supply Max	55 V
Peak Current	+/- 16 A
Maximum Continuous Current	+/- 10 A
Switching Frequency	16 KHz
Case Operating Temperature	-40^0C to $+85^0$C
Voltage drop across bridge at Max Current (and Maximum Temperature)	1.92 V
+/- 15 Outputs Max Current	20 mA
Bandwidth	1.8 MHz

Table A3.1: Amplifier Specifications

APPENDIX A4: MSK 4360 Motor Specifications for the Maxon EC 32 and EC 40

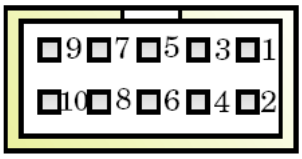
Specification	*Units*	*EC 32 (Hip)*	*EC 40 (Knee)*
Assigned power rating	W	80	120
Nominal voltage	Volt	48	48
No load speed	rpm	11300	10600
Stall Torque	mNm	350	1224
No load current	mA	150	222
Terminal resistance phase to phase	Ohm	5.50	1.69
Max. permissible speed	rpm	25000	18000
Max. continuous current at 5000 rpm	A	1.60	2.90
Max. continuous torque at 5000 rpm	mNm	53.1	112.3
Max efficiency	%	76	84
Torque constant	mNm/A	40.0	43.0
Speed constant	rpm/V	239	222
Mechanical time constant	ms	6.9	8
Terminal inductance phase to phase	mH	0.856	0.460
Thermal time constant winding	s	16	16
Thermal resistance winding-housing	K/W	2.5	1.2
Thermal resistance housing-ambient	K/W	5.4	3.2

Table A4.1: Maxon Motor Specifications

MSK Amp. Pin Name	*Motor Wire Name*
Motor Drive A	Winding 1
Motor Drive B	Winding 2
Motor Drive C	Winding 3
+15 V 20mA Output	V_{Hall} (4.5-24 VDC)
SIG GND	GND
Hall Input 1	Hall Sensor 1
Hall Input 2	Hall Sensor 2
Hall Input 3	Hall Sensor 3

Table A4.2: Motor to Amplifier Connections

APPENDIX A5: KoreMotor Encoder Connections for HP HED 5540, Gurley R112, and Unimeasure JX-EP



1. Motor +	6. Motor –
2. Encoder 5V	7. Motor +
3. Encoder Canal A	8. Motor –
4. Encoder Canal B	9. Encoder Index
5. Encoder Gnd	10. Encoder Index

Figure A5.1: KoreMotor Connector Pinout

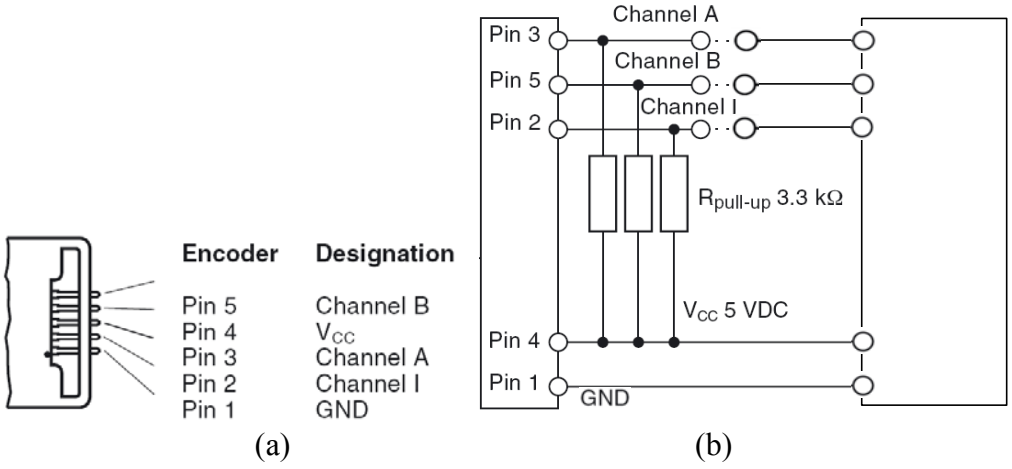


Figure A5.2: (a) HP HED 5540 Pin Locations, and (b) Connections with KoreMotor

*Note: The Gurley R112 and Unimeasure JX-EP have been fitted with connectors that connect directly to a KoreMotor Port. There is no need for pull-up resistors, and the connections are not shown.

APPENDIX A6: K-Team Embedded System Assembly

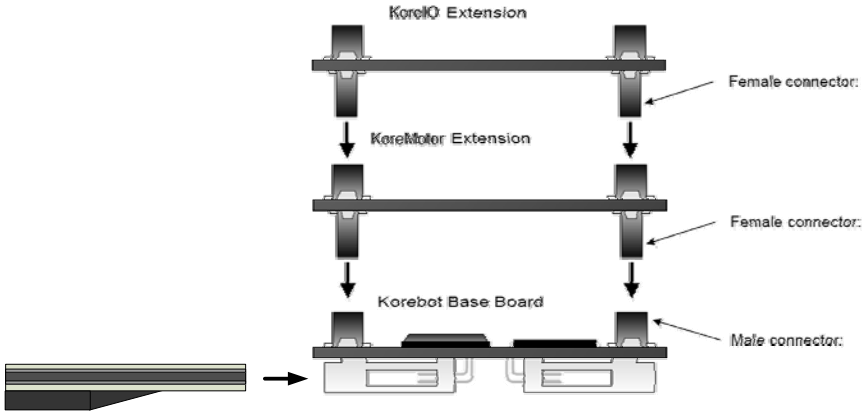


Figure A6.1: K-Team Wireless Card and Peripheral Board Connections

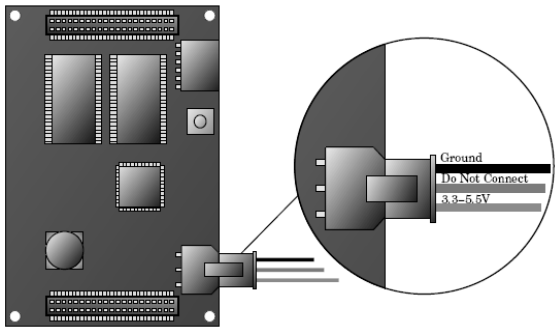


Figure A6.2: KoreBot Power Connection

(a) (b)

Figure A6.3: (a) Documented and (b) Recommended Inter-board Connection

***Note:** There are two issues worth mentioning about the K-Team boards. First, early work by Adam Porr documents that the high voltage (5.5 - 30VDC) power converter was rendered inoperative in early testing by a short. The problem is most likely with the DC to DC converter that steps the high voltage down to the internal 3.3 volts. The low voltage input (3.3 - 5.5VDC) is fully operational. Second, when the KoreMotor board was connected to the KoreBot it was found to not work from time to time. The problem was diagnosed to be a flaky connection that resided in the KoreMotor's KB-250 module. In particular, the connection is broken if the two boards are stacked together as showed in Figure 6.3 (a). A work around is to pull the two boards apart slightly as seen in Figure 6.3 (b).

APPENDIX A7: Interface Circuitry Connections

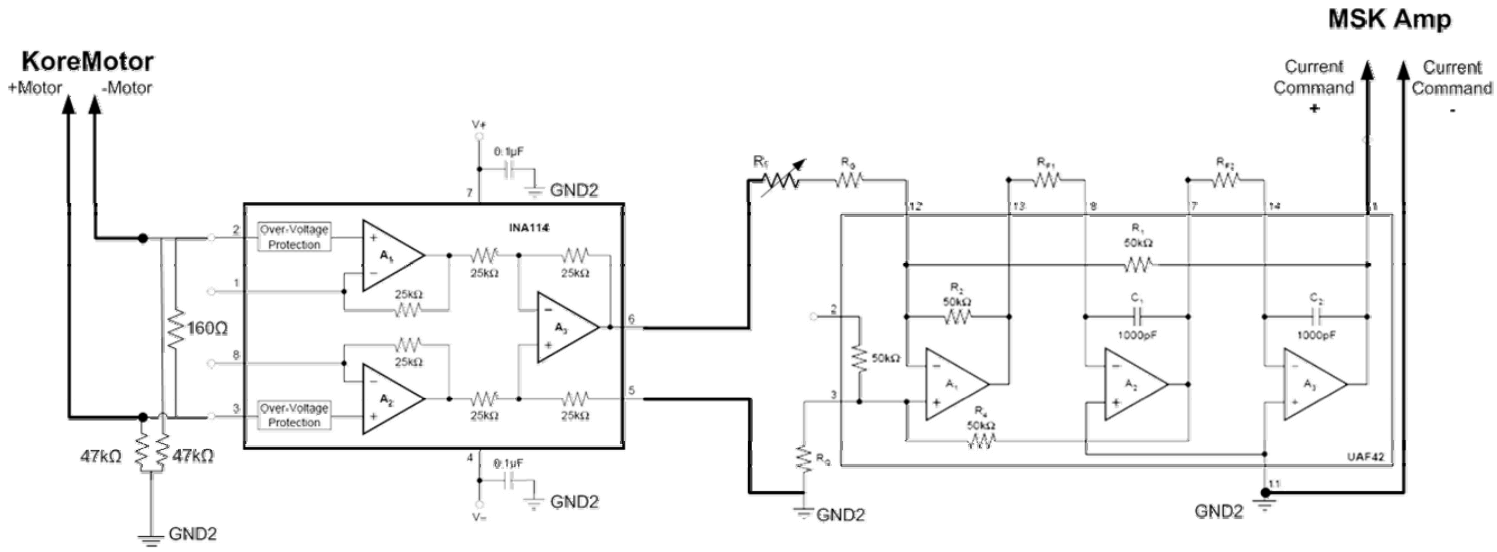


Figure A7.1: Wiring Diagram for Interface Circuitry (single axis).

Resistor	*UAF42 (HIP)*	*UAF42 (KNEE)*
$\mathbf{R_G}$	150KΩ	47KΩ
$\mathbf{R_f}$	100KΩ	10KΩ
$\mathbf{R_{F1}}$	560KΩ	560KΩ
$\mathbf{R_{F2}}$	560KΩ	560KΩ

Table A7.1: Resistor Values for UAF42 Universal Filter.

*Note: With the exception of some resistors, the interface circuit of Figure A7.1 is identical for both axes.

APPENDIX A8: Power Supply and DC to DC Converters

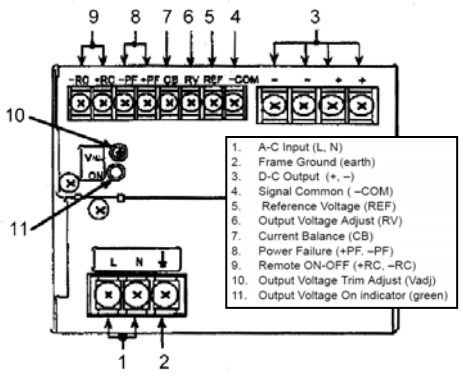


Figure A8.1: Kepco RKE 900W Power Supply Module Interface

Table A8.1: Specifications for Kepco RKE 900W Power Supply

Specification	*Range*
Input Voltage	100-120 VAC
Output Power	900 W
Output Voltage	+48 VDC
Switching Freq.	135 KHz

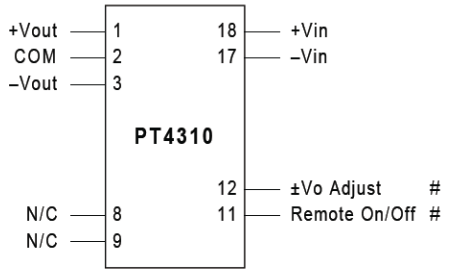


Figure A8.2: PT4310 Module Interface

Table A8.2: Specifications for PT4310 DC to DC Converter

Specification	*Range*
Input Voltage	30 – 75 VDC
Output Power	6 W
Output Voltage	+/-12 VDC
Switching Freq.	500KHz
Turn-On Current	0.1 Amp
Weight	10 grams

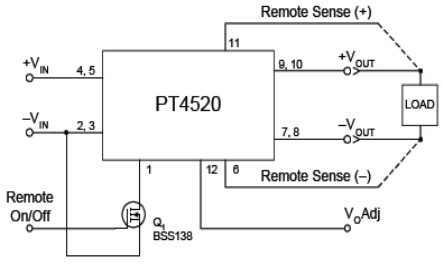


Figure A8.3: PT4520 Module Interface

Table A8.3: Specifications for PT4520 DC to DC Converter

Specification	*Range*
Input Voltage	30 – 75 VDC
Output Power	20 W
Output Voltage	+5 VDC
Switching Freq.	650 KHz
Turn-On Current	0.1 Amp
Weight	23 grams

APPENDIX A9: Chapter 1 of Adam Porr's Hopper Project

Chapter 1. Setting Up the Korebot Development Computer

Abstract

This chapter will focus on how to prepare a laptop or desktop computer to communicate with the Korebot single board computer and develop software for the Korebot platform.

1. Known issues and other warnings

- The Redhat 9 standard kernel WILL crash if you try to use USB networking with the current desktop computer (Dell Precision 220).
- Slackware 10 has a tendency to "freeze" temporarily every now and then, so if you can't move the cursor or type, just wait 10 sec. or so and it should correct itself.
- The current desktop computer has a nasty habit of rebooting sporadically for no apparent reason, so save your work often! This seems to be a Linux-specific problem as I have experienced it under both Redhat and Slackware, but not under Windows.
- Copies of all of the current configuration files that are mentioned below are kept in the */mnt/data/hopper_files/config_files/desktop* directory.

2. Setting up local filesystems

The current desktop computer (Dell Precision 220) contains 2 hard disks. One has a NTFS partition that contains Windows 2000. The other is split into native Linux filesystems and a FAT32 partition for data that is common to Windows and Linux (since Windows can't read EXT3 Linux partitions and Linux can't write to NTFS partitions). This FAT partition, /dev/hdb4 under Linux, is a good place to store any files that you would like to be able to read in Windows. To set it up to mount automatically when Linux boots, add the following line to the */etc/fstab* file:

```
/dev/hdb4        /mnt/data        vfat        users,quiet,rw,umask=000 1 0
```

After you do this, you can mount the drive without restarting by typing **mount /mnt/data**. Once the partition is mounted, you can access it just like any other directory by typing **cd /mnt/data**.

3. Configuring peripherals

4. Setting up Ethernet networking

To configure Ethernet networking under Slackware, simply modify the */etc/rc.d/rc.inet1.conf* file to contain the following:

```
# Config information for eth0:
IPADDR[0]="164.107.163.63"
NETMASK[0]="255.255.252.0"
USE_DHCP[0]="no"
DHCP_HOSTNAME[0]=""
```

Further on in the file:

```
# Default gateway IP address:
GATEWAY="164.107.160.1"
```

If you need to set up another Ethernet interface (to talk to a Compact Flash card on the Korebot, for instance), simply configure the information for eth1 to the relevant parameters.

You'll also have to modify */etc/resolv.conf* to contain the following lines:

```
nameserver 128.146.1.7
nameserver 128.146.48.7
```

These settings will take effect automatically on the next boot. To make them take effect sooner, simply type **/etc/rc.d/rc.inet1**.

5. Setting up USB networking

USB networking is achieved using the usbnet.o Linux kernel module. To use it, the usbnet module must be installed, and the interface must be configured appropriate settings.

The usbnet module is most likely included as part of your Linux distribution. If it is not, you will get the following message when you try to install the module in the running kernel:

```
insmod: usbnet: no module by that name found
```

If this happens, you will need to recompile the kernel modules for whatever kernel version you are using. Doing this is beyond the scope of this document, but I'll be happy to help you if you have trouble. Assuming that the module is present, you should see the following when you type **insmod usbnet** at the command prompt (You will have to be root to do this):

```
Using /lib/modules/2.4.26/kernel/drivers/usb/usbnet.o.gz
```

You will probably need to configure the hotplug subsystem to create the network interface automatically whenever the USB cable and the Korebot are connected to the KoreConnect. Assuming you have hotplug enabled, this can be accomplished in Slackware by creating a file called */etc/hotplug/usb/usbnet* that contains the following text:

```
#!/bin/bash

typeset -i num
num=`ifconfig | grep usb0 | wc -l`
if [ $num -eq 0 ] ; then
ifconfig usb0 192.168.1.1 netmask 255.255.255.0 up
route add -host 192.168.1.2 usb0
fi
```

6. Setting up a NFS filesystem

K-Team suggests creating a Network Filesystem (NFS) share on the desktop computer to allow you to do all of your Korebot software development remotely on the desktop computer. Doing this has two benefits. You can compile your programs on the desktop computer, which is likely substantially faster than the Korebot. Likewise, the ability to access the desktop computer's hard drive remotely from the Korebot eliminates the need to copy your new binary over to the Korebot each time you recompile.

Setting up NFS is easy. First you have to define which directory you want to share and who has permission to access it. This is done in the /etc/exports file. Simply modify the file to contain the following line:

```
/home/hopper     192.168.1.2(rw,insecure,sync)
```

This gives the Korebot (192.168.1.2) permission to mount the */home/hopper* directory remotely. Obviously, you'll have to change it if you change the Korebot's IP address or if you want to remotely access a different directory. I HIGHLY recommend not trying to mount a FAT directory using NFS. I had a lot of trouble early on and I couldn't figure out why, but in retrospect, it was probably due to the fact that I was trying to use NFS with the FAT hard drive.

You will have to tell NFS services about the new mountpoint before you try to mount the NFS drive. To do this, simply type **exportfs -ra**. If NFS isn't running, you will probably get an error message. If this happens, type **/etc/rc.d/rc.nfsd restart** (using Slackware) to (re)start NFS services. If you're still having trouble, check out the NFS-HOWTO at http://nfs.sourceforge.net/nfs-howto/.

7. Installing Korebot development tools and libraries

This section in particular is not documented very well in the K-Team Korebot manual, so I'll try to cover it in depth here. Creating software for the Korebot requires two main components. The development toolchain allows you to crosscompile programs for the Korebot processor using the desktop computer. The Korebot code libraries, known as *libkorebot*, provide functions that can be used to access the various capabilities of the Korebot. K-Team makes it a point to stress that the best way to develop software for the Korebot is to write and compile it on the desktop computer and run it remotely from the Korebot until the final product is ready.

Alternatively, the development toolchain and library can be installed directly on the Korebot. I have not tried this, nor have I really read much of K-Team's documentation about it, so you're on your own if you want to try it. Keep in mind that there is not a lot of extra space for development software on the Korebot filesystem, and the Korebot processor is most likely slower than the processor on the desktop computer.

The first step should be to install the development toolchain. I had a lot of trouble with this because the way K-Team configured the Makefile conflicted with the way they named some of the files that were used in the build process. However, with the current releases of the toolchain and the Linux kernel (korebot-tools-0.1.2 and linux-2.4.19-kb9, respectively), this issue should be resolved. If you use the 2.4.19-kb8.1 version of the kernel, be aware that you will have to change the names of several files in order for the build script to work. The most up-to-date versions of these packages are available from the K-Team website, but there are also a copies stored locally in */mnt/data/hopper_files/software/korebot-tools/* and */mnt/data/hopper_files/software/kernel/*.

To build the toolchain, do the following things:

1. Copy the korebot-tools-*-tar.gz file to your home directory and unzip and untar it. This will create a korebot-tools-* directory.
2. Copy the kernel tar.gz file to the *korebot-tools-*/src/* directory. Don't unpack it - the korebot-tools build script needs it in the tar.gz form.
3. Change to the korebot-tools-*/src directory.
4. Determine your gcc version by typing **gcc -v**.
5. Determine your glibc version by typing **ls /var/log/packages/ | grep glibc**. (Note that this will only work in Slackware)
6. Determine your binutils version by typing **ls /var/log/packages/ | grep binutils** (Note that this will only work in Slackware)
7. Open the *korebot-tools-*/src/build-toolchain* script in your favorite text editor and edit the first few lines so that they look like this:

```
#!/bin/sh -e
PREFIX=`pwd`
PREFIX=${PREFIX%/src}
KERNEL=`pwd`/linux
BINUTILS_VERSION=2.12.1
GCC_VERSION=2.95.3
GLIBC_VERSION=2.2.5
# use the prefix for whatever kernel version you copied to the
# /korebot-tools*/src directory
KERNEL_VERSION=2.4.19-kb9
```

   I was able to successfully compile korebot-tools without modifying any of these version numbers. The important parameter is the kernel version because **build-toolchain** uses this parameter to unpack the kernel archive. Modify the KERNEL_VERSION variable to reflect the name of the kernel archive that you copied into the */korebot-tools*/src* directory. I do not know how crucial it is that the kernel version matches the kernel that is running on the Korebot. It might be important, but I successfully compiled and ran a test program using versions that did not match.

18. Run the script by typing **./build-toolchain**. Be aware that it takes a LONG TIME (several hours) to compile the toolchain. I recommend that you let it go overnight.
19. Copy the korebot-tools-* directory containing the compiled tools to a convenient location. It doesn't really matter where as long as you compiled the toolchain from source. (If you use the K-Team binary, you'll need to follow their instructions on where to put it. I didn't have much luck with their binary, but you might try it before you try to compile your own.)

When the tools have finished compiling, you'll need to add the location of the newly compiled binaries to your path. The best way to do this is to append the path to the directory to the PATH environment variable in the */etc/profile* file so that it looks like this:

```
PATH="/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/games:/home/hopper/korebot-tools-0.1.2/bin"
```

Note that you will probably have to exit X-Windows and logout in order for the new PATH to take effect. Alternatively, you can use the following command to set the PATH temporarily until your next login:

```
echo PATH=$PATH:/home/hopper/korebot-tools-0.1.2/bin
```

Once you have added the directory to your PATH, you can test it with the command **arm-linux-gcc -v**, which should yield the following output:

```
Reading specs from /home/hopper/korebot-tools-0.1.2/lib/gcc-lib/arm-linux/2.95.3/specs
gcc version 2.95.3 20010315 (release)
```

The next step should be to install the Korebot libraries. There is not much to this. The most up-to-date version is available from the K-Team website, but there is also a copy stored locally in */mnt/data/hopper_files/software/libkorebot/*. Regardless of where you get it from, copy the tar.gz file to your home directory and unzip it. Then change to the unzipped directory and type make. This will build the header files and shared libraries that your programs will use. When the compilation is finished, change to the *build-korebot/include* directory and copy all of the header files to a convenient place. You'll probably want to put them in the directory that will be mounted using NFS so that you can easily copy them to the Korebot later on. Next, change to the *build-korebot/lib* directory and copy the shared library files to a convenient place. A safe bet would be to put them in the same directory where you put the header files.

APPENDIX A10: LibKoreBot API Commands

Function and Procedure Call	Description	Time Delay (ms)
kmot_GetMeasure – position (t_{RP})	Returns Encoder Position	1.80
kmot_GetMeasure – speed (t_{RS})	Returns Encoder Speed	0.90
kmot_SetPoint (t_{IC})	Issues Control Command	2.37
kmot_SetSampleTime	Sets Control Step Time	0.64
kmot_SetPosition	Sets Encoder Values	1.28
kmot_ConfigurePID	Sets PID Coefficients	1.90
kmot_SetSpeedProfile	Sets Max Speed And Acceleration	0.96
kmot_SetLimits	Sets Speed and Position Limits	2.56
kmot_SetMode	Change µC's Mode of Operation	0.32
kmot_SetMargin	Set Margin for Desired Position	0.32
kmot_SetOptions	Set µC's Option Register	0.64
kmot_ResetError	Clear µC's Error Register	0.32
kmot_SetBlockedTime	Set Time Motor Can Be Blocked	0.32
kmot_GetStatus	Get Status Flags µC	0.90
kio_ReadIO	Read Digital Input from IO	0.47

Table A10.1: LibKoreBot API Commands with Time Delays

APPENDIX A11: C Code for Entire Console Application

```
/*-------------------------------------------------------------------
* hopper.c - HONORS THESIS - Simon Curran, Dr. Orin
*-------------------------------------------------------------------
* This console application is meant to be run from a secure terminal
* on the K-Team KoreBot.
*-------------------------------------------------------------------
* $Author: S. Curran $
* $Date: 7/01/2005 08:42:05 $
* $Revision: 21 $
*-------------------------------------------------------------------*/

#include <signal.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <korebot/korebot.h>
#include <time.h>
#include <sys/time.h>

 struct timeval start_time, end_time;

/*!
* \file   hopper.c is used to control the Hopping Leg
*
* \brief
*         kmot_test provides a small utility program to communicate with the
*         the KoreMotor and send commands to the motor controllers. Use the
*         help command to get a list of available controls.
*
*
* \author   Simon Curran (The Ohio State University)
*
* \bug      none discovered.
* \todo     nothing.
*/
static int quitReq = 0;
static int stopReq = 0;

static knet_dev_t * hip;
static knet_dev_t * knee;
static knet_dev_t * stringPot;
static knet_dev_t * gurley;
static knet_dev_t * koreio;
static knet_dev_t * tempMotor;


struct timeb { time_t time;
                                                        unsigned short millitm;
                                                        short timezone;
                                                        short dstflag;
                                        };

/*!
* Make sure the program terminate properly on a ctrl-c
*/
static void ctrlc_handler( int sig )
{
 stopReq = 1;
}

/*-------------------------------------------------------------------*/
/*! Quit the program.
*/
int quit( int argc , char * argv[] )
{
 quitReq = 1;
}

/*-------------------------------------------------------------------*/
/*! Stop the current motor (set mode to stop motor mode).
*/
int stop( int argc , char * argv[] )
{
 kmot_SetMode( hip , kMotModeStopMotor );
 kmot_SetMode( knee , kMotModeStopMotor );
 kmot_SetMode( stringPot , kMotModeStopMotor );
 kmot_SetMode( gurley , kMotModeStopMotor );
}


/*-------------------------------------------------------------------*/
/*! Initialize all the parameters for the current controller. This command
* must be called before using any other command, and the parameters should
* be modified if there is a replacement of the hip/knee motors or encoders.
*/
int init( int argc , char * argv[] )
```

```
{

 //Activates controllers
 kmot_SetMode( hip , kMotModeIdle );
 kmot_SetMode( knee , kMotModeIdle );

 kmot_SetSampleTime( hip , 1550 );  //2.48ms
 kmot_SetSampleTime( knee , 1550 );  //2.48ms
 kmot_SetSampleTime( stringPot , 1550 );  //2.48ms
 kmot_SetSampleTime( gurley , 1550 );  //2.48ms
 //kmot_SetSampleTime( knee , 768 );  //1.2288ms
 //kmot_SetSampleTime( knee , 768 );  //1.2288ms

 kmot_SetMargin( hip , 20 );
 kmot_SetMargin( knee , 20 );

 kmot_SetOptions( hip ,
                  0x0 ,
                  kMotSWOptWindup | kMotSWOptStopMotorBlk );
 kmot_SetOptions( knee ,
                  0x0 ,
                  kMotSWOptWindup | kMotSWOptStopMotorBlk );

 kmot_ResetError( hip );
 kmot_ResetError( knee );
 kmot_ResetError( stringPot );
 kmot_ResetError( gurley );

 kmot_SetBlockedTime( hip , 10 );
 kmot_SetBlockedTime( knee , 10 );
 kmot_SetLimits( hip , kMotRegCurrent , 0 , 500 );
 kmot_SetLimits( knee , kMotRegCurrent , 0 , 500 );
 kmot_SetLimits( hip , kMotRegPos , -10000 , 10000 );
 kmot_SetLimits( knee , kMotRegPos , -10000 , 10000 );

 /* PID  */
 kmot_ConfigurePID(hip,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(knee,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(hip,kMotRegPos, 34, 180, 0 );
 kmot_ConfigurePID(knee,kMotRegPos, 7, 60, 0 );

 kmot_SetSpeedProfile(hip,-30,-10);  //speed, accel
 kmot_SetSpeedProfile(knee,-30,-10);  //speed, accel
 return 0;
}


/*---------------------------------------------------------------------*/
/*! Set the PID controller gains for the HIP motor.
* syntax: setpid <regulation type> <Kp> <Ki> <Kd>
*
* regulation types are: pos, posprofile, speed, speedprofile, and torque.
*/
int setpidHip( int argc , char * argv[] )
{

 int regType[] = {
   kMotRegPos ,
   kMotRegPosProfile ,
   kMotRegSpeed ,
   kMotRegSpeedProfile ,
   kMotRegTorque
 };

 char * regTypeStr[] = {
   "pos" ,
   "posprofile" ,
   "speed" ,
   "speedprofile" ,
   "torque" ,
   NULL
 };

 int reg, kp, ki, kd;

 for (reg=0; regTypeStr[reg] != NULL; reg++) {
   if (!strcasecmp( argv[1] , regTypeStr[reg] )) {

     kp = atoi(argv[2]);
     ki = atoi(argv[3]);
     kd = atoi(argv[4]);

     printf("Set (P,I,D) to (%d,%d,%d)\n" , kp , ki , kd );
     kmot_ConfigurePID( hip , regType[reg] , kp , kd , ki );
   }
 }
}

/*---------------------------------------------------------------------*/
/*! Set the PID controller gains for the KNEE motor.
* syntax: setpid <regulation type> <Kp> <Ki> <Kd>
*
* regulation types are: pos, posprofile, speed, speedprofile, and torque.
*/
int setpidKnee( int argc , char * argv[] )
```

```
{

 int regType[] = {
   kMotRegPos ,
   kMotRegPosProfile ,
   kMotRegSpeed ,
   kMotRegSpeedProfile ,
   kMotRegTorque
 };

 char * regTypeStr[] = {
   "pos" ,
   "posprofile" ,
   "speed" ,
   "speedprofile" ,
   "torque" ,
   NULL
 };

 int reg, kp, ki, kd;

 for (reg=0; regTypeStr[reg] != NULL; reg++) {
   if (!strcasecmp( argv[1] , regTypeStr[reg] )) {

     kp = atoi(argv[2]);
     ki = atoi(argv[3]);
     kd = atoi(argv[4]);

     printf("Set (P,I,D) to (%d,%d,%d)\n" , kp , ki , kd );
     kmot_ConfigurePID( knee , regType[reg] , kp , kd , ki );
   }
 }
}

/*-----------------------------------------------------------------------*/
/*! Set a new position regulation target for the current motor. The
* controller must be properly initialized before using regulation.
* syntax: setpos <target position>
*/
int setpos( int argc , char * argv[] )
{
FILE *outfile1;
FILE *outfile2;
int v[3];

kmot_ConfigurePID(hip,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(knee,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(hip,kMotRegPos, 34, 180, 1 );
 kmot_ConfigurePID(knee,kMotRegPos, 7, 60, 0 );

//opens local output files and stores data on KoreBot
if ( (outfile1 = fopen( "/home/hopper/pos.dat","w")) == NULL)
          {
                    printf("Can't open %s\n","pos.dat");
                    return(1);
          }
if ( (outfile2 = fopen( "/home/hopper/vel.dat","w")) == NULL)
          {
                    printf("Can't open %s\n","vel.dat");
                    return(1);
          }

 //Resets Encoders to zero
 kmot_SetPosition(knee, 0);
 kmot_SetPosition(hip, 0);

 //Commands the desired position using Closed Loop Regulation on pos.
 kmot_SetPoint( knee , 1, atoi(argv[2]) );
 kmot_SetPoint( hip , 1, atoi(argv[1]));

 //Collects data until CNTRL+C is pressed
  while(!stopReq)
          {
          v[0] = kmot_GetMeasure( hip , kMotMesPos );
          v[1] = kmot_GetMeasure( knee , kMotMesPos );
          v[2] = kmot_GetMeasure( knee , kMotMesCurrent );
          fprintf(outfile1, "%d\n", v[0]);
          fprintf(outfile2, "%d\n", v[1]);
          }

//Commands zero current
kmot_SetPoint( knee , 0 , 0);
kmot_SetPoint( hip , 0 , 0);
fclose(outfile1);
fclose(outfile2);

//UNCOMMENT TO COPY AUTOMATICALLY COPY FILES TO DEVELOPMENT PC (/mnt/nfs)
//system("cp -f /home/hopper/pos.dat /mnt/nfs/v7/");
//system("cp -f /home/hopper/vel.dat /mnt/nfs/v7/");
 stopReq = 0;
 return 0;

}
```

```
/*---------------------------------------------------------------------*/
/* There is no inital physical configuration required to run this.
   Simply type kick, and stand back.  Watch to findout where to place
   a ball if you want.
*/
int kick( int argc , char * argv[] )
{
kmot_ConfigurePID(hip,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(knee,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(hip,kMotRegPos, 34, 180, 1 );
 kmot_ConfigurePID(knee,kMotRegPos, 7, 60, 0 );

kmot_SetPoint( knee , 0, -6000 );
usleep(1000000);
usleep(1000000);
usleep(500000);

 kmot_SetPosition(knee, 0);
 kmot_SetPosition(hip, 0);

 kmot_SetPoint( knee , 1, 77000 );
 usleep(80000 );
 kmot_SetPoint( hip , 1, -15000 );

 usleep(1000000);

 kmot_SetPoint( knee , 1, 45000 );
 //kmot_SetPoint( hip , 1, -5000 );
 usleep(1000000);
 kmot_SetPoint( knee , 1, 55000 );
 kmot_SetPoint( hip , 1, -5000 );
 usleep(1000000);

kmot_SetPoint( knee , 0 , 0);
kmot_SetPoint( hip , 0 , 0);

 stopReq = 0;
 return 0;

}
```

```
/*--------------------------------------------------------------------*/
/* This is the final hopping routine.  It is documented in Simon Curran's
   Honor's Thesis in Chapter 4.  The usage is a follows:
   hop <+HipCurrent> <+KneeCurrent> <uS spent Crouching>
   Example:  hop 32767 32767 50000
                              This will provide both the hip and knee with full current in
                              the correct direction and spend 50ms effectively falling into
                              a crouch.  Note, that because the larger gear box the Knee is
                              actually is provided a small negative current to help get it
                              out of the way.

                              For example it would return something like:
                              Out of Crouch at exactly: 50020 uS
                              Broke Contact at: 205001 uS
*/
int hop( int argc , char * argv[] )
{
//Defines all local helper variables
int hipCurrent = atoi(argv[1]);
int kneeCurrent = atoi(argv[2]);
int max_time = 350000+(atoi(argv[3])); //crouch time
FILE *outfile1;
FILE *outfile2;
int h[10000];
int k[10000];
int i = 0;
int final_i = 0;
int total_usecs = 0;
int contact[10];
int contact_sum = 5;
char *junk;
char buf2[32];
contact[0] = 0;
contact[1] = 0;
contact[2] = 0;
contact[3] = 0;
contact[4] = 0;
contact[5] = 0;
contact[6] = 0;
contact[7] = 0;
contact[8] = 0;
contact[9] = 0;

//Opens Two Output Files, Can be Used for Generic Purposes.
if ( (outfile1 = fopen( "/home/hopper/pos.dat","w")) == NULL)
          {
                    printf("Can't open %s\n","pos.dat");
                    return(1);
          }
if ( (outfile2 = fopen( "/home/hopper/vel.dat","w")) == NULL)
          {
                    printf("Can't open %s\n","vel.dat");
                    return(1);
          }

 //STANCE STATE
 //Zeros out the encoder values
 kmot_SetPosition(hip, 0);
 kmot_SetPosition(knee, 0);
 kmot_SetPosition(stringPot, 0);
 //Commands the Leg to hold in place, maintaining stance.
 kmot_SetPoint( knee , 1, 0 );
 kmot_SetPoint( hip , 1, 0 );
 //Awaits user keypress.
 junk = fgets(buf2, 2, stdin);  //Transition to next state

 //CROUCH STATE
 //turns off Hip motor an retracts Knee
 kmot_SetPoint( knee , 0, -5000 );
 kmot_SetPoint( hip , 0, 0 );

 //start High Resolution Timer
 gettimeofday(&start_time, NULL);

          //Get Elapsed Time Value in total_usecs
          gettimeofday(&end_time, NULL);
          total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 + (end_time.tv_usec-start_time.tv_usec);

          //Perform iterations until elapsed time exceeds Crouch State's time
          while((total_usecs < (atoi(argv[3]))) && !(stopReq))
          {
                    //Check Time
                    gettimeofday(&end_time, NULL);
                    total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 + (end_time.tv_usec-
start_time.tv_usec);

                    //Request only 1 measurement, height
                    h[i] = kmot_GetMeasure( stringPot , kMotMesPos );
                    i = i+1;  //increment data collection counter
          }
          //transitioning out of Crouch State, time limit reached.
          printf("Out of Crouch at exactly: %i uS\n", total_usecs); //informs user of exact transition time
```

```
  //THRUSTING STATE
  if(!stopReq)
  {
          //Send open loop currents to motors unless CNTRL+C was pressed
                  kmot_SetPoint( knee , 0, kneeCurrent );
                  kmot_SetPoint( hip , 0, hipCurrent );
  }
  i = i + 1; //increment data collection counter

  //performs iterations until the 3ms block of foot sensor data is all zero indicating
  //that the foot is no longer in contact with the ground.  Effectively, debouncing it.
  while ((contact_sum > 0) && (!stopReq) && (total_usecs < (max_time + atoi(argv[3]))))
  {
          //Samples the foot sensor in 6 consecutive samples (Each .5 ms apart, total = 3ms)
          contact[0] = kio_ReadIO(koreio,1);
          contact[1] = kio_ReadIO(koreio,1);
          contact[2] = kio_ReadIO(koreio,1);
          contact[3] = kio_ReadIO(koreio,1);
          contact[4] = kio_ReadIO(koreio,1);
          contact[5] = kio_ReadIO(koreio,1);
          h[i] = kmot_GetMeasure( stringPot , kMotMesPos );  //captures height data

          contact_sum = contact[0] + contact[1] + contact[2] + contact[3] + contact[4] + contact[5];


          gettimeofday(&end_time, NULL);  //checks current time
          total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 + (end_time.tv_usec-start_time.tv_usec);
          i = i+1;  //increments data collection counter
  }
  //Transitions out of Thrust State b/c contact is broken
  printf("Broke Contact at: %i uS\n", total_usecs); //informs user at what time foot left ground.


  //FLIGHT / TOUCHDOWN STATE
  //these commands catch the leg and
  //reposition it for touchdown. To be optimized.
  kmot_SetPoint( knee , 1, 9600 );
  kmot_SetPoint( hip , 1, 4200 );

  //program stays in final state until user presses CNTRL+C
  //Data collection continues on height only.
  while(!stopReq)
  {
         h[i] = kmot_GetMeasure( stringPot , kMotMesPos );
         i = i+1;
  }

//CNTRL+C was pressed, command zero current to both motors
kmot_SetPoint( hip , 0 , 0);
kmot_SetPoint( knee , 0 , 0);

//Writes data collection array to file(s)
final_i = i;
i = 0;
while((i <= final_i))
{
          fprintf(outfile1, "%i\n", h[i]);
          fprintf(outfile2, "%i\n", k[i]);
          i = i + 1;
}

//Closes files
fclose(outfile1);
fclose(outfile2);
//Copies Files To Development System (/mnt/nfs corresponding to /home/hopper on PC)
system("cp -f /home/hopper/pos.dat /mnt/nfs/v7/");
system("cp -f /home/hopper/vel.dat /mnt/nfs/v7/");
 stopReq = 0;
 return 0;
}


/*---------------------------------------------------------------------*/
/*! Get a set of measures from the current hip. This control returns
* the speed, position and current for the hip.
*/
int measure( int argc , char * argv[] )
{
 int p[4];
 int v[4];

 //capture raw position encoder data
 p[0] = kmot_GetMeasure( hip , kMotMesPos );
 p[1] = kmot_GetMeasure( knee , kMotMesPos );
 p[2] = kmot_GetMeasure( stringPot , kMotMesPos );
 p[3] = kmot_GetMeasure( gurley , kMotMesPos );

 //capture raw speed encoder data
 v[0] = kmot_GetMeasure( hip , kMotMesSpeed );
 v[1] = kmot_GetMeasure( knee , kMotMesSpeed );
 v[3] = kmot_GetMeasure( knee , kMotMesSpeed );
 v[4] = kmot_GetMeasure( knee , kMotMesSpeed );

 //display values on screen
printf( "posHip: %d, speedHip: %d\n" ,
```

```
         p[0] , v[0]);
printf( "posKnee: %d, speedKnee: %d\n" ,
         p[1] , v[1]);
printf( "posStringPot: %d, speedStringPot: %d\n" ,
         p[2] , v[2]);
printf( "posGurley: %d, speedGurley: %d\n" ,
         p[3] , v[3]);
 return 0;
}

int hold(int argc , char * argv[])
{
 int hipPos = 0;
 int kneePos = 0;

 //for safety, reset controller PID values for each axis using
 //experimentally tuned PID gains.
 kmot_ConfigurePID(hip,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(knee,kMotRegSpeed , 0 , 0 , 0 );
 kmot_ConfigurePID(hip,kMotRegPos, 34, 180, 1 );
 kmot_ConfigurePID(knee,kMotRegPos, 7, 60, 0 );

 //obtain current positions for hip and knee
 hipPos = kmot_GetMeasure( hip , kMotMesPos );
 kneePos = kmot_GetMeasure( knee , kMotMesPos );

 //command both HIP and KNEE to do PID control about current positions
 kmot_SetPoint( knee , 1, kneePos );
 kmot_SetPoint( hip , 1, hipPos );

 while(!stopReq)  //wait until CNTRL+C is pressed
 {}

 //zeros both HIP and KNEE current commands
 kmot_SetPoint( knee , 0, 0 );
 kmot_SetPoint( hip , 0, 0 );

stopReq = 0; //resets CNTRL+C flag
return 0;
}


int AutoCalibrate(int argc , char * argv[])
{
          //openloop current commands drive both axis towards hardstops
          kmot_SetPoint( knee , 0 , -6000);
          kmot_SetPoint( hip , 0 , -15000);

          //sufficent delay added to allow contact w/ hardstops as well
          //as let all impact vibrations die out.
          usleep(1000000);
          usleep(500000);

          //Set Hip and Knee encoder counts to zero
          kmot_SetPosition( knee , 0);
          kmot_SetPosition( hip , 0);

          //Sets gurley and stringpot Encoder counts to a predetermined and
          //known value corresponding to a physical characteristic
          kmot_SetPosition( stringPot , 0);
          kmot_SetPosition( gurley , 0);

          //Kills openloop current command
          kmot_SetPoint( knee , 0 , 0);
          kmot_SetPoint( hip , 0 , 0);
          return 0;
}

int clear( int argc , char * argv[] )
{
//Commands zero current to the controller,
//effectively clearling all commands
//zeros all Encoder data.
kmot_SetPoint( knee , 0 , 0);
kmot_SetPoint( hip , 0 , 0);
kmot_SetPosition( hip , 0);
kmot_SetPosition( knee , 0);
kmot_SetPosition( stringPot , 0);
kmot_SetPosition( gurley , 0);
}


/*
Commands OPEN LOOP current to the two axis.
Argument 1 is the motor number, A '1' commands the HIP, '2' the KNEE
Argument 2 is the commanded current value
The commanded value has range [-32767 32767] and is none linear within this region.
Knee Current Range [-10 Amps 10 Amps]
Hip Current Range [-3.6 Amps 3.6 Amps]
*/
int openloop( int argc , char * argv[] )
{
int p[1];
int v[1];
```

```
          if (atoi(argv[1]) == 1) //if '1' command HIP
          {

          //kill any commands running in HIP
          kmot_SetPoint( hip , 0 , 0);

          //Perform until CNTRL+C is pressed
          while(!stopReq )
          {
                    //Command open loop current from argument 2
                    kmot_SetPoint( hip , 0 , atoi(argv[2]));
                     //capture raw position encoder data
 p[0] = kmot_GetMeasure( hip , kMotMesPos );

 //capture raw speed encoder data
 v[0] = kmot_GetMeasure( hip , kMotMesSpeed );

                    printf( "posHip: %d, speedHip: %d\n" ,
         p[0] , v[0]);
          }

          //commands zero current to HIP
         kmot_SetPoint( hip , 0 , 0);
        }

        else //command KNEE
        {
         //kill any commands running in KNEE
         kmot_SetPoint( knee , 0 , 0);

          //Perform until CNTRL+C is pressed
          while(!stopReq )
          {
                    //Command open loop current from argument 2
                    kmot_SetPoint( knee , 0 , atoi(argv[2]));
          }

          //commands zero current to KNEE
         kmot_SetPoint( knee , 0 , 0);
        }

        //reset CNTRL + C flag
        stopReq = 0;
        //exit function
        return 0;
}

/*-----------------------------------------------------------------------*/
/*! Reset the error register of the motor controllers.
*/
int statusclear( int argc , char * argv[] )
{
 kmot_ResetError(hip);
 kmot_ResetError(knee);
 kmot_ResetError(stringPot);
 kmot_ResetError(gurley);
}

/*-----------------------------------------------------------------------*/
/*! Read and print the status of the current controller. The content of
* the status register and error register will be displayed.
*/
int status( int argc , char * argv[] )
{
 unsigned char error , status;

 kmot_GetStatus( hip , &status , &error );

 printf( "status=%02X error=%02X\n" ,
         status , error );

 if ( status & kMotStatusMoveDet )
   printf( "Movement detect!\n" );

 printf( "Direction %s !\n" ,
            (status&kMotStatusDir) ? "Negative" : "Positive" );

 if ( status & kMotStatusOnSetPt )
   printf( "On Set Point!\n" );

 if ( status & kMotStatusNearSetPt )
   printf( "Near Set Point !\n" );

 if ( status & kMotStatusCmdSat )
   printf( "Command saturated !\n" );

 if ( status & kMotStatusWindup )
   printf( "Antireset Wind up active !\n" );

 if ( status & kMotStatusSoftCurCtrl )
   printf( "Software Current Control Active !\n" );

 if ( status & kMotStatusSoftStop )
   printf( "Software Stop Active !\n" );
```

```
 if ( error & kMotErrorSampleTimeTooSmall )
   printf( "Sample Time Too Small !\n" );

 if ( error & kMotErrorWDTOverflow )
   printf( "WatchDot Timer Overflow !\n" );

 if ( error & kMotErrorBrownOut )
   printf( "Brown-out !\n" );

 if ( error & kMotErrorSoftStopMotor )
   printf( "Software Stopped Motor !\n" );

 if ( error & kMotErrorMotorBlocked )
   printf( "Motor Blocked !\n" );

 if ( error & kMotErrorPosOutOfRange )
   printf( "Position Out of Range !\n" );

 if ( error & kMotErrorSpeedOutOfRange )
   printf( "Speed Out of Range !\n" );

 if ( error & kMotErrorTorqueOutOfRange )
   printf( "Torque Out of Range !\n" );

 return 0;
}

int help( int argc , char * argv[] );
/*-------------------------------------------------------------------------*/
/*! The command table contains:
* command name : min number of args : max number of args : the function to call
*/
static kb_command_t cmds[] = {
 { "quit"            , 0 , 0 , quit } ,
 { "clear"            , 0 , 0 , clear } ,
 { "exit"            , 0 , 0 , quit } ,
 { "bye"             , 0 , 0 , quit } ,
 { "hold"             , 0 , 0 , hold } ,
 { "stop"            , 0 , 0 , stop } ,
 { "kick"            , 0 , 0 , kick } ,
 { "init"            , 0 , 0 , init } ,
 { "setpidHip"       , 4 , 4 , setpidHip } ,
 { "setpidKnee"      , 4 , 4 , setpidKnee } ,
 { "setpos"          , 2 , 2 , setpos } ,
 { "measure"         , 0 , 0 , measure } ,
 { "status"          , 0 , 0 , status } ,
 { "statusclear"     , 0 , 0 , statusclear } ,
 { "setcurrent"      , 2 , 2 , openloop } ,
 { "calibrate"       , 0 , 0 , AutoCalibrate } ,
 { "hop"                , 3 , 3 , hop } ,
 { "help"            , 0 , 0 , help } ,
 { NULL              , 0 , 0 , NULL }
};

/*-------------------------------------------------------------------------*/
/*! Display a list of available commands.
*/
int help( int argc , char * argv[] )
{
 kb_command_t * scan = cmds;
 while(scan->name != NULL)
 {
   printf("%s\r\n",scan->name);
   scan++;
 }
 return 0;
}

/*-------------------------------------------------------------------------*/
/*! Main program to process the command line.
*
*/
static char buf[1024];
static char buf2[64];

int main( int argc , char * argv[] )
{
 int rc, ver;
 char * name1;
 char * name2;
 char * name3;
 char * name4;

 //Names are undocumented literals from K-Team
 //the motor numbers are off by +1 compared w/ K-Team documentation.
 //Where PriMotor1 refers to motor port 0.
 //PriMotor refers to having the Motor Board's dip switches set to i2c address
 //range 1.  Address range two would be "AltMotor" instead of "PriMotor."
 name1 = "KoreMotor:PriMotor1";
 name2 = "KoreMotor:PriMotor2";
 name3 = "KoreMotor:PriMotor3";
 name4 = "KoreMotor:PriMotor4";


 /* Set the libkorebot debug level - Highly recommended for development. */
```

```
kb_set_debug_level(2);

//if returns < 0, there is a problem with the KoreBot, program is terminated.
if ((rc = kb_init( argc , argv )) < 0 )
  return 1;

//captures any software inturrupt (CNTRL + C) and
//calls function ctrlc_handler.
signal( SIGINT , ctrlc_handler );

 //open KoreIO board and requet device pointer for 'koreio'
 //Dip switch on IO board set to address range 1
 koreio = knet_open( "KoreIO:Board", KNET_BUS_ANY, 0 , NULL );

 if(!koreio) //if IO boards fails to open, try address range 2
 {
   printf("Cannot open KoreIO device trying alternate address\r\n");
   koreio = knet_open( "KoreIO:AltBoard", KNET_BUS_ANY, 0 , NULL );
   if(!koreio) //if address range 2 fails, terminate program.
   {
     printf("Cannot open KoreIO device\r\n");
     return 1;
   }
 }

 /* Get and display the koreio firmware version */
 kio_GetFWVersion(koreio,&ver);

 printf("KoreIO firmware %d.%d\r\n", (ver&0x000000F0)>>4, (ver&0x0000000F));

//open all four motor ports on the Motor Board and request device
//pointers for each.
hip = knet_open( name1 , KNET_BUS_I2C , 0 , NULL );
knee = knet_open( name2 , KNET_BUS_I2C , 0 , NULL );
gurley = knet_open( name3 , KNET_BUS_I2C , 0 , NULL );
stringPot = knet_open( name4 , KNET_BUS_I2C , 0 , NULL );


//if they all open sucessfully, print the Firmware version # for each PIC
if ( hip && knee && stringPot && gurley )
{
  unsigned int ver , rev;
  unsigned char status , error;
  int min, max;
  int junk;

  kmot_GetFWVersion( hip , &ver );

  printf("Hip Motor Firmware v%u.%u\n" ,
      KMOT_VERSION(ver) , KMOT_REVISION(ver) );

  kmot_GetFWVersion( knee , &ver );

  printf("Knee Motor Firmware v%u.%u\n" ,
      KMOT_VERSION(ver) , KMOT_REVISION(ver) );

  kmot_GetFWVersion( stringPot , &ver );

  printf("Motor Port for String Pot Firmware v%u.%u\n" ,
      KMOT_VERSION(ver) , KMOT_REVISION(ver) );

  kmot_GetFWVersion( gurley , &ver );

  printf("Motor Port for Gurley v%u.%u\n" ,
      KMOT_VERSION(ver) , KMOT_REVISION(ver) );

  junk = init(0, 0);

  //Create a Command Prompt interface using K-Team parser to interpret input
  //All comands are cross checked with cmds[] array by the parser.
  while (!quitReq) {
    printf("\n> ");
    if ( fgets( buf , sizeof(buf) , stdin ) != NULL ) {
      buf[strlen(buf)-1] = '\0';
      kb_parse_command( buf , cmds );
    }


  }

  //release I2C control of all motor ports as well as KoreIO
  knet_close( hip );
  knet_close( knee );
  knet_close( stringPot );
  knet_close( gurley );
  knet_close( koreio );
}
else
{
  printf("Cannot open KoreMotor device(s)\r\n");
}

return 0;
}
```

APPENDIX A12: Ziegler Nichols Method for Tuning the PID Controller

First, note that the required proportional control gain is positive. That is the steady state process gain was found to be positive and the required proportional control gain, K_p, is positive as well.

1. Turn the controller to P-only mode, i.e. turn both the Integral and Derivative modes off.

2. Turn the controller gain, K_p, up slowly (in the positive direction) and observe the output response. Note that this requires changing K_p in step increments and waiting for a steady state in the output, before another change in K_p is implemented.

3. When a value of K_p results in a sustained periodic oscillation in the output (or close to it), make note of this critical value of K_p.

4. Reduce K_p slightly and add in a small derivative gain, K_d.

5. Turn the controller gain K_d, up slowly (in the positive direction) and observe the output response. Again note, this requires changing K_d in step increments and waiting for a steady state in the output, before another change in K_d is implemented.

6. When a value of K_d results in a non oscillatory output with the desired amount of damping, assign this as the final value for K_d.

7. If so desired, the integral gain, K_i, can be turned on to improve the steady state error and more perfectly track the desired command.

8. This value should be slowly increased, usually only in increments of one. Observe the output and wait for it to reach steady state. When the needed accuracy is obtained assign this as the final value for K_i. There is a chance that steps 5 through 6 will need to be repeated to damp out any instability that has come about.

REFERENCES

[1] J. M. Remic III, "Prototype Leg Design for a Quadruped Robot Application," M.S. Thesis, Department of Mechanical Engineering, The Ohio State University, June 2005.

[2] M. H . Raibert, "Legged Robots That Balance." MIT Press, Cambridge, Mass., 1996.

[3] G. A. Pratt and M. M. Williamson, "Series Elastic Actuators," M.S. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1995

[4] R. B. McGhee, D. E. Orin, D. R. Pugh, and M. R. Patterson, "A hierarchically-structured system for computer control of a hexapod walking machine," in *Theory and Practice of Robots and Manipulators, Proceedings of RoManSy-84 Symposium* (A. Morecki, ed.), (London), pp. 375-381, Hermes Publishing Co., 1985.

[5] D. Pugh, E. Ribble, V. Vohnout, T. Bihari, T. Walliser, M. Patterson, and K. Waldron, "Technical description of the Adaptive Suspension Vehicle," *The International Journal on Robotics Research,* vol. 9, pp. 24-42, April 1990.

[6] K. S. Espenschied, R. D. Quinn, H. J. Chiel, and R. D. Beer, "Leg coordination mechanisms in the stick insect applied to hexapod robot locomotion," *Adaptive Behavior,* no. 4, pp. 455-468, 1993.

[7] K. S. Espenschied, R. D. Quinn, H. J. Chiel, and R. D. Beer, "Biologically based distributed control and local reflexes improve rough terrain locomotion in a hexapod robot," *Robotics and Autonomous Systems,* pp. 59-64, 1996.

[8] J. A. Smith and I. Poulakakis, "Rotary gallop in the untethered quadrupedal robot Scout II," in Proceedings of the International Conference on Intelligent Robots and Systems (IROS) 2004, (Sendai, Japan), pp. 2556–2561, 2004.

[9] D. P. Krasny and D. E. Orin, "Evolution of a 3D Gallop in a Quadrupedal Robot Using a Simple Energy Control Approach," submitted to *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics,* June 2005.

[10] R. M. Alexander, A. S. Jayes, and R. F. Ker, "Estimates of energy cost for quadrupedal running gaits," Journal of Zoology, vol. 190, pp. 155–192, 1980.

[11] P. Nanua and K. J. Waldron, "Energy comparison between trot, bound, and gallop using a simple model," ASME Journal of Biomechanical Engineering, vol. 117, pp. 466–473, 1995.

[12] C. T. Rubin and L. E. Lanyon, "Limb mechanics as a function of speed and gait: A study of functional strains in the radius and tibia of horse and dog," Journal of Experimental Biology, vol. 101, pp. 187–211, 1982.

[13] A. A. Biewener and C. R. Taylor, "Bone strain: A determinant of gait and speed," Journal of Experimental Biology, vol. 123, pp. 383–400, 1986.

[14] C. T. Farley and C. R. Taylor, “A mechanical trigger for the trot-gallop transition in horses,” Science, vol. 253, pp. 306–308, 1991.

[15] J. P. Schmiedeler, The Mechanics of and Robotic Design for Quadrupedal Galloping. PhD thesis, The Ohio State University, Columbus, Ohio, 2001.

[16] M. H. Raibert, “Trotting, pacing, and bounding by a quadruped robot,” *Journal of Biomechanics,* vol. 23, pp. 79-98, 1990.

[17] J. G. Nichol, S. P. N. Singh, K. J. Waldron, L. R. Palmer III, and D. E. Orin, "System Design of a Quadrupedal Galloping Machine," *The International Journal of Robotics Research*, vol. 23, no. 10-11, pp. 1013-1027, October-November 2004.

[18] D. P. Krasny, and D. E. Orin, “Generating High-Speed Dynamic Running Gaits in a Quadruped Robot Using an Evolutionary Search,” in *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, Vol. 34, No. 4, pp. 1685-96, August 2004.

[19] D. P. Krasny, and D. E. Orin, “Achieving Periodic Leg Trajectories to Evolve a Quadruped Gallop,” in *Proc. of the 2003 IEEE Intl. Conference on Robotics & Automation,* (Taipei, Taiwan), pp. 3842-8, September 14-19.

[20] L. R. Palmer, D. E. Orin, D. W. Marhefka, J. P. Schmiedeler, and K. J. Waldron, "Intelligent Control of an Experimental Articulated Leg for a Galloping Machine," in *Proc. of the 2003 IEEE Intl. Conference on Robotics & Automation,* (Taipei, Taiwan), pp. 3821-7, September 14-19.